Note to readers:
Please ignore these
sidenotes; they're just
hints to myself for
preparing the index,
and they're often flaky!

KNUTH

THE ART OF COMPUTER PROGRAMMING

VOLUME 4 PRE-FASCICLE 5B

INTRODUCTION TO BACKTRACKING

DONALD E. KNUTH *Stanford University*

ADDISON–WESLEY

December 21, 2015

Internet
Stanford GraphBase
MMIX

Internet page `http://www-cs-faculty.stanford.edu/~knuth/taocp.html` contains current information about this book and related books.

See also `http://www-cs-faculty.stanford.edu/~knuth/sgb.html` for information about *The Stanford GraphBase*, including downloadable software for dealing with the graphs used in many of the examples in Chapter 7.

See also `http://www-cs-faculty.stanford.edu/~knuth/mmixware.html` for downloadable software to simulate the `MMIX` computer.

Copyright © 2015 by Addison–Wesley

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher, except that the official electronic file may be used to print single copies for personal (not commercial) use.

Zeroth printing (revision -86), 18 December 2015

CARROLL
Internet

PREFACE

Begin at the beginning, *and do not allow yourself to gratify*
a mere idle curiosity by dipping into the book, here and there.
This would very likely lead to your throwing it aside,
with the remark "This is much *too hard for me!,"*
and thus losing the chance of adding a very large *item*
to your stock of mental delights.

— LEWIS CARROLL, in *Symbolic Logic* (1896)

THIS BOOKLET contains draft material that I'm circulating to experts in the field, in hopes that they can help remove its most egregious errors before too many other people see it. I am also, however, posting it on the Internet for courageous and/or random readers who don't mind the risk of reading a few pages that have not yet reached a very mature state. *Beware:* This material has not yet been proofread as thoroughly as the manuscripts of Volumes 1, 2, 3, and 4A were at the time of their first printings. And those carefully-checked volumes, alas, were subsequently found to contain thousands of mistakes.

Given this caveat, I hope that my errors this time will not be so numerous and/or obtrusive that you will be discouraged from reading the material carefully. I did try to make the text both interesting and authoritative, as far as it goes. But the field is vast; I cannot hope to have surrounded it enough to corral it completely. So I beg you to let me know about any deficiencies that you discover.

To put the material in context, this portion of fascicle 5 previews the opening pages of Section 7.2.2 of *The Art of Computer Programming*, entitled "Backtrack programming." The preceding section, 7.2.1, was about "Generating basic combinatorial patterns" — namely tuples, permutations, combinations, partitions, and trees. Now it's time to consider the non-basic patterns, the ones that have a much less uniform structure. For these we generally need to make tentative choices and then we need to back up when those choices need revision. Several subsections (7.2.2.1, 7.2.2.2, etc.) will follow this introductory material.

* * *

The explosion of research in combinatorial algorithms since the 1970s has meant that I cannot hope to be aware of all the important ideas in this field. I've tried my best to get the story right, yet I fear that in many respects I'm woefully ignorant. So I beg expert readers to steer me in appropriate directions.

Please look, for example, at the exercises that I've classed as research problems (rated with difficulty level 46 or higher), namely exercises 14, ...; I've also implicitly mentioned or posed additional unsolved questions in the answers to exercises 6, 8, 42, 45, ... Are those problems still open? Please inform me if you know of a solution to any of these intriguing questions. And of course if no solution is known today but you do make progress on any of them in the future, I hope you'll let me know.

I urgently need your help also with respect to some exercises that I made up as I was preparing this material. I certainly don't like to receive credit for things that have already been published by others, and most of these results are quite natural "fruits" that were just waiting to be "plucked." Therefore please tell me if you know who deserves to be credited, with respect to the ideas found in exercises 31(b), 33, 44, 50, 51, 95, ... Furthermore I've credited exercises ... to unpublished work of .... Have any of those results ever appeared in print, to your knowledge?

I've got a historical question too: Have you any idea who originated the idea of "stamping" in data structures? (See 7.2.2–(26). This concept is quite different from the so-called time stamps in persistent data structures, and quite different from the so-called time stamps in depth-first search algorithms, and quite different from the so-called time stamps in cryptology, although many programmers do use the name "time stamp" for those kinds of stamp.) It's a technique that I've seen often, in programs that have come to my attention during recent decades, but I wonder if it ever appeared in a book or paper that was published before, say, 1980.

stamping
Knuth

* * *

Special thanks are due to ... for their detailed comments on my early attempts at exposition, as well as to numerous other correspondents who have contributed crucial corrections.

* * *

I happily offer a "finder's fee" of $2.56 for each error in this draft when it is first reported to me, whether that error be typographical, technical, or historical. The same reward holds for items that I forgot to put in the index. And valuable suggestions for improvements to the text are worth 32¢ each. (Furthermore, if you find a better solution to an exercise, I'll actually do my best to give you immortal glory, by publishing your name in the eventual book:–)

Cross references to yet-unwritten material sometimes appear as '00'; this impossible value is a placeholder for the actual numbers to be supplied later.

Happy reading!

Stanford, California D. E. K.
99 Umbruary 2015

Part of the Preface to Volume 4B

During the years that I've been preparing Volume 4, I've often run across basic techniques of probability theory that I would have put into Section 1.2 of Volume 1 if I'd been clairvoyant enough to anticipate them in the 1960s. Finally I realized that I ought to collect most of them together in one place, near the beginning of Volume 4B, because the story of these developments is too interesting to be broken up into little pieces scattered here and there.

Therefore this volume begins with a special section entitled "Mathematical Preliminaries Redux," and future sections use the abbreviation 'MPR' to refer to its equations and its exercises.

MATHEMATICAL PRELIMINARIES REDUX

Many parts of this book deal with *discrete probabilities*, namely with a finite or countably infinite set Ω of atomic events ω, each of which has a given probability $\Pr(\omega)$, where

$$0 \le \Pr(\omega) \le 1 \qquad \text{and} \qquad \sum_{\omega \in \Omega} \Pr(\omega) = 1. \tag{1}$$

. . .

For the complete text of the special MPR section, please see Pre-Fascicle 5a. Incidentally, Section 7.2.2 intentionally begins on a left-hand page, and its illustrations are numbered beginning with Fig. 68, because Section 7.2.1 ended on a right-hand page and its final illustration was Fig. 67. The editor has decided to treat Chapter 7 as a single unit, even though it will be split across several physical volumes.

KING
LENNON
backtrack
Walker
domain
cutoff
properties: logical propositions (relations)
$P_0()$
lexicographically
n queens–

Nowhere to go but out,
Nowhere to come but back.
— BEN KING, in *The Sum of Life* (c. 1893)

No one I think is in my tree.
— JOHN LENNON, in *Strawberry Fields Forever* (1967)

7.2.2. Backtrack Programming

Now that we know how to generate simple combinatorial patterns such as tuples, permutations, combinations, partitions, and trees, we're ready to tackle more exotic patterns that have subtler and less uniform structure. Instances of almost *any* desired pattern can be generated systematically, at least in principle, if we organize the search carefully. Such a method was christened "backtrack" by R. J. Walker in the 1950s, because it is basically a way to examine all fruitful possibilities while exiting gracefully from situations that have been fully explored.

Most of the patterns we shall deal with can be cast in a simple, general framework: We seek all sequences $x_1x_2\ldots x_n$ for which some property $P_n(x_1,x_2,\ldots,x_n)$ holds, where each item x_k belongs to some given domain D_k of integers. The backtrack method, in its most elementary form, consists of inventing intermediate "cutoff" properties $P_l(x_1,\ldots,x_l)$ for $1 \le l < n$, such that

$$P_l(x_1,\ldots,x_l) \text{ is true whenever } P_{l+1}(x_1,\ldots,x_{l+1}) \text{ is true;} \qquad (1)$$

$$P_l(x_1,\ldots,x_l) \text{ is fairly easy to test, if } P_{l-1}(x_1,\ldots,x_{l-1}) \text{ holds.} \qquad (2)$$

(We assume that $P_0()$ is always true. Exercise 1 shows that all of the basic patterns studied in Section 7.2.1 can easily be formulated in terms of domains D_k and cutoff properties P_l.) Then we can proceed lexicographically as follows:

Algorithm B (*Basic backtrack*). Given domains D_k and properties P_l as above, this algorithm visits all sequences $x_1x_2\ldots x_n$ that satisfy $P_n(x_1,x_2,\ldots,x_n)$.

B1. [Initialize.] Set $l \leftarrow 1$, and initialize the data structures needed later.

B2. [Enter level l.] (Now $P_{l-1}(x_1,\ldots,x_{l-1})$ holds.) If $l > n$, visit $x_1x_2\ldots x_n$ and go to B5. Otherwise set $x_l \leftarrow \min D_l$, the smallest element of D_l.

B3. [Try x_l.] If $P_l(x_1,\ldots,x_l)$ holds, update the data structures to facilitate testing P_{l+1}, set $l \leftarrow l+1$, and go to B2.

B4. [Try again.] If $x_l \ne \max D_l$, set x_l to the next larger element of D_l and return to B3.

B5. [Backtrack.] Set $l \leftarrow l-1$. If $l > 0$, downdate the data structures by undoing the changes recently made in step B3, and return to B4. (Otherwise stop.) ▮

The main point is that if $P_l(x_1,\ldots,x_l)$ is false in step B3, we needn't waste time trying to append any further values $x_{l+1}\ldots x_n$. Thus we can often rule out huge regions of the space of all potential solutions. A second important point is that very little memory is needed, although there may be many, many solutions.

For example, let's consider the classic *problem of n queens*: In how many ways can n queens be placed on an $n \times n$ board so that no two are in the same

diagonal
backtrack tree
profile

row, column, or diagonal? We can suppose that one queen is in each row, and that the queen in row k is in column x_k, for $1 \le k \le n$. Then each domain D_k is $\{1, 2, \ldots, n\}$; and $P_n(x_1, \ldots, x_n)$ is the condition that

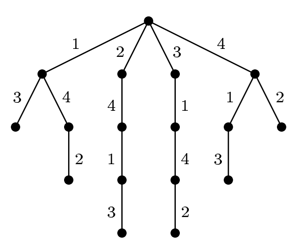
$$x_j \ne x_k \quad \text{and} \quad |x_k - x_j| \ne k - j, \qquad \text{for } 1 \le j < k \le n. \tag{3}$$

(If $x_j = x_k$ and $j < k$, two queens are in the same column; if $|x_k - x_j| = k - j$, they're in the same diagonal.)

This problem is easy to set up for Algorithm B, because we can let property $P_l(x_1, \ldots, x_l)$ be the same as (3) but restricted to $1 \le j < k \le l$. Condition (1) is clear; and so is condition (2), because P_l requires testing (3) only for $k = l$ when P_{l-1} is known. Notice that $P_1(x_1)$ is always true in this example.

One of the best ways to learn about backtracking is to execute Algorithm B by hand in the special case $n = 4$ of the n queens problem: First we set $x_1 \leftarrow 1$. Then when $l = 2$ we find $P_2(1,1)$ and $P_2(1,2)$ false; hence we don't get to $l = 3$ until trying $x_2 \leftarrow 3$. Then, however, we're stuck, because $P_3(1,3,x)$ is false for $1 \le x \le 4$. Backtracking to level 2, we now try $x_2 \leftarrow 4$; and this allows us to set $x_3 \leftarrow 2$. However, we're stuck again, at level 4; and this time we must back up all the way to level 1, because there are no further valid choices at levels 3 and 2. The next choice $x_1 \leftarrow 2$ does, happily, lead to a solution without much further ado, namely $x_1x_2x_3x_4 = 2413$. And one more solution (3142) turns up before the algorithm terminates.

The behavior of Algorithm B is nicely visualized as a tree structure, called a search tree or *backtrack tree*. For example, the backtrack tree for the four queens problem has just 17 nodes,



, (4)

corresponding to the 17 times step B2 is performed. Here x_l is shown as the label of an edge from level $l-1$ to level l of the tree. (Level l of the algorithm actually corresponds to the tree's level $l-1$, because we've chosen to represent patterns using subscripts from 1 to n instead of from 0 to $n-1$ in this discussion.) The *profile* $(p_0, p_1, \ldots, p_n)$ of this particular tree — the number of nodes at each level — is $(1,4,6,4,2)$; and we see that the number of solutions, $p_n = p_4$, is 2.

Figure 68 shows the corresponding tree when $n = 8$. This tree has 2057 nodes, distributed according to the profile $(1, 8, 42, 140, 344, 568, 550, 312, 92)$. Thus the early cutoffs facilitated by backtracking have allowed us to find all 92 solutions by examine only 0.01% of the $8^8 = 16{,}777{,}216$ possible sequences $x_1 \ldots x_8$. (And 8^8 is only 0.38% of the $\binom{64}{8} = 4{,}426{,}165{,}368$ ways to put eight queens on the board.)

data structures–
mems
downdating vs updating+
undoes

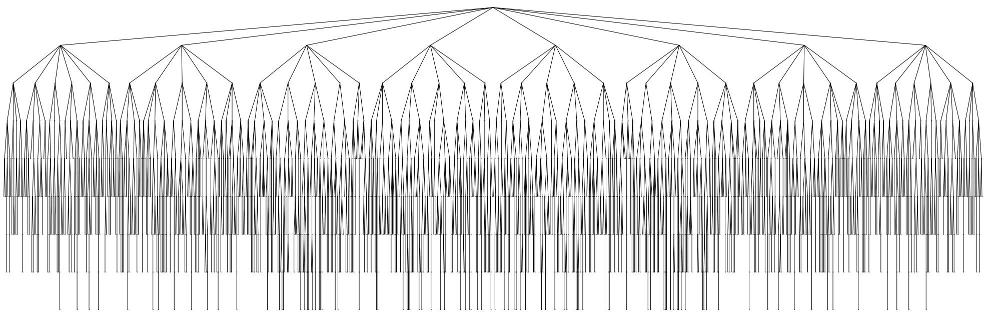


Fig. 68. The problem of placing eight nonattacking queens has this backtrack tree.

Notice that, in this case, Algorithm B spends most of its time in the vicinity of level 5. Such behavior is typical: The backtrack tree for $n = 16$ queens has 1,141,190,303 nodes, and its profile is (1, 16, 210, 2236, 19688, 141812, 838816, 3998456, 15324708, 46358876, 108478966, 193892860, 260303408, 253897632, 171158018, 72002088, 14772512), concentrated near level 12.

Data structures. Backtrack programming is often used when a *huge* tree of possibilities needs to be examined. Thus we want to be able to test property P_l as quickly as possible in step B3.

One way to implement Algorithm B for the n queens problem is to avoid auxiliary data structures and simply to make a bunch of sequential comparisons in that step: "Is $x_l - x_j \in \{j-l, 0, l-j\}$ for some $j < l$?" Assuming that we access memory whenever referring to x_j, given a trial value x_l in a register, such an implementation performs approximately 112 billion memory accesses when $n = 16$; that's about 98 mems per node.

We can do better by introducing three simple arrays. Property P_l in (3) says essentially that the numbers x_k are distinct, and so are the numbers $x_k + k$, and so are the numbers $x_k - k$. Therefore we can use auxiliary Boolean arrays $a_1 \ldots a_n$, $b_1 \ldots b_{2n-1}$, and $c_1 \ldots c_{2n-1}$, where a_j means 'some $x_k = j$', b_j means 'some $x_k + k - 1 = j$', and c_j means 'some $x_k - k + n = j$'. Those arrays are readily updated and downdated if we customize Algorithm B as follows:

B1*. [Initialize.] Set $a_1 \ldots a_n \leftarrow 0 \ldots 0$, $b_1 \ldots b_{2n-1} \leftarrow 0 \ldots 0$, $c_1 \ldots c_{2n-1} \leftarrow 0 \ldots 0$, and $l \leftarrow 1$.

B2*. [Enter level l.] (Now $P_{l-1}(x_1, \ldots, x_{l-1})$ holds.) If $l > n$, visit $x_1 x_2 \ldots x_n$ and go to B5*. Otherwise set $t \leftarrow 1$.

B3*. [Try t.] If $a_t = 1$ or $b_{t+l-1} = 1$ or $c_{t-l+n} = 1$, go to B4*. Otherwise set $a_t \leftarrow 1$, $b_{t+l-1} \leftarrow 1$, $c_{t-l+n} \leftarrow 1$, $x_l \leftarrow t$, $l \leftarrow l + 1$, and go to B2*.

B4*. [Try again.] If $t < n$, set $t \leftarrow t + 1$ and return to B3*.

B5*. [Backtrack.] Set $l \leftarrow l - 1$. If $l > 0$, set $t \leftarrow x_l$, $c_{t-l+n} \leftarrow 0$, $b_{t+l-1} \leftarrow 0$, $a_t \leftarrow 0$, and return to B4*. (Otherwise stop.) ▮

Notice how step B5* neatly undoes the updates that step B3* had made, in the reverse order. Reverse order for downdating is typical of backtrack algorithms,

although there is some flexibility; we could, for example, have restored a_t before b_{t+l-1} and c_{t-l+n}, because those arrays are independent.

The auxiliary arrays a, b, c make it easy to test property P_l at the beginning of step B3*, but we must also access memory when we update them and downdate them. Does that cost us more than it saves? Fortunately, no: The running time for $n = 16$ goes down to about 34 billion mems, roughly 30 mems per node.

Furthermore we could keep the bit vectors a, b, c entirely in registers, on a machine with 64-bit registers, assuming that $n \le 32$. Then there would be just two memory accesses per node, namely to store $x_l \leftarrow t$ and later to fetch $t \leftarrow x_l$; however, quite a lot of in-register computation would become necessary.

registers
Walker
cutoff properties
visits
bitwise operations
historical notes+
Sprague

Walker's method. The 1950s-era programs of R. J. Walker organized backtracking in a somewhat different way. Instead of letting x_l run through all elements of D_l, he calculated and stored the set

$$S_l \leftarrow \{x \in D_l \mid P_l(x_1, \ldots, x_{l-1}, x) \text{ holds}\} \qquad (5)$$

upon entry to each node at level l. This computation can often be done efficiently all at once, instead of piecemeal, because some cutoff properties make it possible to combine steps that would otherwise have to be repeated for each $x \in D_l$. In essence, he used the following variant of Algorithm B:

Algorithm W (*Walker's backtrack*). Given domains D_k and cutoffs P_l as above, this algorithm visits all sequences $x_1x_2\ldots x_n$ that satisfy $P_n(x_1, x_2, \ldots, x_n)$.

W1. [Initialize.] Set $l \leftarrow 1$, and initialize the data structures needed later.

W2. [Enter level l.] (Now $P_{l-1}(x_1, \ldots, x_{l-1})$ holds.) If $l > n$, visit $x_1x_2\ldots x_n$ and go to W4. Otherwise determine the set S_l as in (5).

W3. [Try to advance.] If S_l is nonempty, set $x_l \leftarrow \min S_l$, update the data structures to facilitate computing S_{l+1}, set $l \leftarrow l + 1$, and go to W2.

W4. [Backtrack.] Set $l \leftarrow l - 1$. If $l > 0$, downdate the data structures by undoing changes made in step W3, set $S_l \leftarrow S_l \setminus x_l$, and retreat to W3. ▮

Walker applied this method to the n queens problem by computing $S_l = U \setminus A_l \setminus B_l \setminus C_l$, where $U = D_l = \{1, \ldots, n\}$ and

$$A_l = \{x_j \mid 1 \le j < l\},\ B_l = \{x_j + j - l \mid 1 \le j < l\},\ C_l = \{x_j - j + l \mid 1 \le j < l\}. \quad (6)$$

He represented these auxiliary sets by bit vectors a, b, c, analogous to (but different from) the bit vectors of Algorithm B* above. Exercise 9 shows that the updating in step W3 is easy, using bitwise operations on n-bit numbers; furthermore, no downdating is needed in step W4. The corresponding run time when $n = 16$ turns out to be just 9.1 gigamems, or 8 mems per node.

Let $Q(n)$ be the number of solutions to the n queens problem. Then we have

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$Q(n)$	1	1	0	0	2	10	4	40	92	352	724	2680	14200	73712	365596	2279184	14772512

and the values for $n \le 11$ were computed independently by several people during the nineteenth century. Small cases were relatively easy; but when T. B. Sprague

had finished computing $Q(11)$ he remarked that "This was a very heavy piece of work, and occupied most of my leisure time for several months. ... It will, I imagine, be scarcely possible to obtain results for larger boards, unless a number of persons co-operate in the work." [See *Proc. Edinburgh Math. Soc.* **17** (1899), 43–68; Sprague was the leading actuary of his day.] Nevertheless, H. Onnen went on to evaluate $Q(12) = 14{,}200$ — an astonishing feat of hand calculation — in 1910. [See W. Ahrens, *Math. Unterhaltungen und Spiele* **2**, second edition (1918), 344.]

All of these hard-won results were confirmed in 1960 by R. J. Walker, using the SWAC computer at UCLA and the method of exercise 9. Walker also computed $Q(13)$; but he couldn't go any further with the machine available to him at the time. The next step, $Q(14)$, was computed by Michael D. Kennedy at the University of Tennessee in 1963, commandeering an IBM 1620 for 120 hours. S. R. Bunch evaluated $Q(15)$ in 1974 at the University of Illinois, using about two hours on an IBM System 360-75; then J. R. Bitner found $Q(16)$ after about three hours on the same computer, but with an improved method.

Computers and algorithms have continued to get better, of course, and such results are now obtained almost instantly. Hence larger and larger values of n lie at the frontier. The current record as of 2015 is $Q(26) = 22{,}317{,}699{,}616{,}364{,}044$, found in 2009 by Thomas B. Preußer of the University of Dresden. (His distributed computation occupied a dynamic cluster of up to 26 diverse FPGA devices for 270 days; those devices provided a total peak of 550 custom-designed hardware solvers to handle 25,204,802 subproblems individually.)

Sprague
Onnen
Ahrens
Walker
SWAC computer
UCLA
Kennedy
University of Tennessee
IBM 1620
Bunch
University of Illinois
IBM System 360-75
Bitner
Preußer
University of Dresden
distributed computation
FPGA
permutation
data structures
Langford pairs
dual
linked list

Permutations and Langford pairs. Every solution $x_1 \ldots x_n$ to the n queens problem is a permutation of $\{1, \ldots, n\}$, and many other problems are permutation-based. Indeed, we've already seen Algorithm 7.2.1.2X, which is an elegant backtrack procedure specifically designed for special kinds of permutations. When that algorithm begins to choose the value of x_l, it makes all of the appropriate elements $\{1, 2, \ldots, n\} \setminus \{x_1, \ldots, x_{l-1}\}$ conveniently accessible in a linked list.

We can get further insight into such data structures by returning to the problem of Langford pairs, which was discussed at the very beginning of Chapter 7. That problem can be reformulated as the task of finding all permutations of $\{1, 2, \ldots, n\} \cup \{-1, -2, \ldots, -n\}$ with the property that

$$x_j = k \quad \text{implies} \quad x_{j+k+1} = -k, \qquad \text{for } 1 \le j \le 2n \text{ and } 1 \le k \le n. \tag{7}$$

For example, when $n = 4$ there are two solutions, namely $234\bar{2}1\bar{3}\bar{1}\bar{4}$ and $413\bar{1}2\bar{4}\bar{3}\bar{2}$. (As usual we find it convenient to write $\bar{1}$ for -1, $\bar{2}$ for -2, etc.) Notice that whenever $x = x_1 x_2 \ldots x_{2n}$ is a solution, its "dual" $-x^R = (-x_{2n}) \ldots (-x_2)(-x_1)$ is also a solution.

Here's a Langford-inspired adaptation of Algorithm 7.2.1.2X, with the former notation modified slightly to match Algorithms B and W: We want to maintain pointers $p_0 p_1 \ldots p_n$ such that, if the positive integers not already present in $x_1 \ldots x_{l-1}$ are $k_1 < k_2 < \cdots < k_t$ when we're choosing x_l, we have the linked list

$$p_0 = k_1,\ p_{k_1} = k_2,\ \ldots,\ p_{k_{t-1}} = k_t,\ p_{k_t} = 0. \tag{8}$$

Such a condition turns out to be easy to maintain.

undoes
deletion operation
dancing links
search tree
cutoff principle

Algorithm L (*Langford pairs*). This algorithm visits all solutions $x_1 \ldots x_{2n}$ to (7) in lexicographic order, using pointers $p_0 p_1 \ldots p_n$ that satisfy (8), and also using an auxiliary array $y_1 \ldots y_{2n}$ for backtracking.

L1. [Initialize.] Set $x_1 \ldots x_{2n} \leftarrow 0 \ldots 0$, $p_k \leftarrow k+1$ for $0 \le k < n$, $p_n \leftarrow 0$, $l \leftarrow 1$.

L2. [Enter level l.] Set $k \leftarrow p_0$. If $k = 0$, visit $x_1 x_2 \ldots x_{2n}$ and go to L5. Otherwise set $j \leftarrow 0$, and while $x_l < 0$ set $l \leftarrow l + 1$.

L3. [Try $x_l = k$.] (At this point we have $k = p_j$.) If $l + k + 1 > 2n$, go to L5. Otherwise, if $x_{l+k+1} = 0$, set $x_l \leftarrow k$, $x_{l+k+1} \leftarrow -k$, $y_l \leftarrow j$, $p_j \leftarrow p_k$, $l \leftarrow l + 1$, and return to L2.

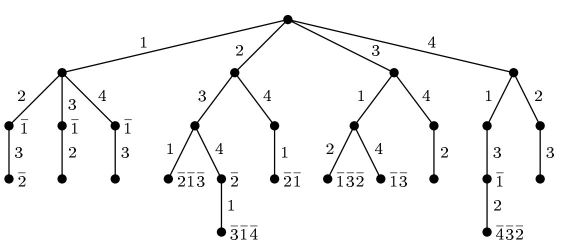
L4. [Try again.] (We've found all solutions that begin with $x_1 \ldots x_{l-1} k$ or something smaller.) Set $j \leftarrow k$ and $k \leftarrow p_j$, then go to L3 if $k \ne 0$.

L5. [Backtrack.] Set $l \leftarrow l - 1$. If $l > 0$ do the following: While $x_l < 0$, set $l \leftarrow l - 1$. Then set $k \leftarrow x_l$, $x_l \leftarrow 0$, $x_{l+k+1} \leftarrow 0$, $j \leftarrow y_l$, $p_j \leftarrow k$, and go back to L4. Otherwise terminate the algorithm. ▌

Careful study of these steps will reveal how everything fits together nicely. Notice that, for example, step L3 removes k from the linked list (8) by simply setting $p_j \leftarrow p_k$. That step also sets $x_{l+k+1} \leftarrow -k$, in accordance with (7), so that we can skip over position $l + k + 1$ when we encounter it later in step L2.

The main point of Algorithm L is the somewhat subtle way in which step L5 undoes the deletion operation by setting $p_j \leftarrow k$. The pointer p_k still retains the appropriate link to the next element in the list, *because p_k has not been changed by any of the intervening updates*. (Think about it.) This is the germ of an idea called "dancing links" that we will explore in Section 7.2.2.1.

To draw the search tree corresponding to a run of Algorithm L, we can label the edges with the positive choices of x_l as we did in (4), while labeling the nodes with any previously set negative values that are passed over in step L2. For instance the tree for $n = 4$ is



. (9)

Solutions appear at depth n in this tree, even though they involve $2n$ values $x_1 x_2 \ldots x_{2n}$.

Algorithm L sometimes makes false starts and doesn't realize the problem until probing further than necessary. Notice that the value $x_l = k$ can appear only when $l + k + 1 \le 2n$; hence if we haven't seen k by the time l reaches $2n - k - 1$, we're *forced* to choose $x_l = k$. For example, the branch $12\bar{1}$ in (9) needn't be pursued, because 4 must appear in $\{x_1, x_2, x_3\}$. Exercise 20 explains how to incorporate this cutoff principle into Algorithm L. When $n = 17$, it reduces the number of nodes in the search tree from 1.29 trillion to 330 billion,

break the symmetry
dual
word rectangles–
footnote doesn't show up here
n-letter words
WORDS
5-letter words
Stanford GraphBase
trie

and reduces the running time from 25.0 teramems to 8.1 teramems. (The amount of work has gone from 19.4 mems per node to 24.4 mems per node, because of the extra tests for cutoffs, yet there's a significant overall reduction.)

Furthermore, we can "break the symmetry" by ensuring that we don't consider both a solution and its dual. This idea, exploited in exercise 21, reduces the search tree to just 160 billion nodes and costs just 3.94 teramems — that's 24.6 mems per node.

Word rectangles. Let's look next at a problem where the search domains D_l are much larger. An $m \times n$ *word rectangle* is an array of n-letter words* whose columns are m-letter words. For example,

```
status
lowest
utopia
making
sledge
```
(10)

is a 5×6 word rectangle whose columns all belong to WORDS(5757), the collection of 5-letter words in the Stanford GraphBase. To find such patterns, we can suppose that column l contains the x_lth most common 5-letter word, where $1 \le x_l \le 5757$ for $1 \le l \le 6$; hence there are $5757^6 = 36{,}406{,}369{,}848{,}837{,}732{,}146{,}649$ ways to choose the columns. In (10) we have $x_1 \ldots x_6 = 1446\ 185\ 1021\ 2537\ 66\ 255$. Of course very few of those choices will yield suitable rows; but backtracking will hopefully help us to find all solutions in a reasonable amount of time.

We can set this problem up for Algorithm B by storing the n-letter words in a trie (see Section 6.3), with one trie node of size 26 for each l-letter prefix of a legitimate word, $0 \le l \le n$.

For example, such a trie for $n = 6$ represents 15727 words with 23667 nodes. The prefix `st` corresponds to node number 260, whose 26 entries are

$$(484,0,0,0,1589,0,0,0,2609,0,0,0,0,0,1280,0,0,251,0,0,563,0,0,0,1621,0); \qquad (11)$$

this means that `sta` is node 484, `ste` is node 1589, ..., `sty` is node 1621, and there are no 6-letter words beginning with `stb`, `stc`, ..., `stx`, `stz`. A slightly different convention is used for prefixes of length $n - 1$; for example, the entries for node 580, '`corne`', are

$$(3879,0,0,3878,0,0,0,0,0,0,0,9602,0,0,0,0,0,171,0,5013,0,0,0,0,0,0), \qquad (12)$$

meaning that `cornea`, `corned`, `cornel`, `corner`, and `cornet` are ranked 3879, 3878, 9602, 171, and 5013 in the list of 6-letter words.

* Whenever five-letter words are used in the examples of this book, they're taken from the 5757 Stanford GraphBase words as explained at the beginning of Chapter 7. Words of other lengths are taken from the *The Official SCRABBLE® Players Dictionary*, fourth edition (Hasbro, 2005), because those words have been incorporated into many widely available computer games. Such words have been ranked according to the British National Corpus of 2007 — where '`the`' occurs 5,405,633 times and the next-most common word, '`of`', occurs roughly half as often (3,021,525). The OSPD4 list includes respectively (101, 1004, 4002, 8887, 15727, 23958, 29718, 29130, 22314) words of lengths (2, 3, ..., 10), of which (97, 771, 2451, 4474, 6910, 8852, 9205, 8225, 6626) occur at least six times in the British National Corpus.

Suppose x_1 and x_2 specify the 5-letter column-words `slums` and `total` as in (10). Then the trie tells us that the next column-word x_3 must have the form $c_1c_2c_3c_4c_5$ where $c_1 \in \{\mathtt{a,e,i,o,r,u,y}\}$, $c_2 \notin \{\mathtt{e,h,j,k,y,z}\}$, $c_3 \in \{\mathtt{e,m,o,t}\}$, $c_4 \notin \{\mathtt{a,b,o}\}$, and $c_5 \in \{\mathtt{a,e,i,o,u,y}\}$. (There are 221 such words.)

Let $a_{l1} \ldots a_{lm}$ be the trie nodes corresponding to the prefixes of the first l columns of a partial solution to the word rectangle problem. This auxiliary array enables Algorithm B to find all solutions, as explained in exercise 24. It turns out that there are exactly 625,415 valid 5×6 word rectangles, according to our conventions; and the method of exercise 24 needs about 19 teramems of computation to find them all. In fact, the profile of the search tree is

$$(1,\ 5757,\ 2458830,\ 360728099,\ 579940198,\ 29621728,\ 625415), \qquad (13)$$

indicating for example that just 360,728,099 of the $5757^3 = 190{,}804{,}533{,}093$ choices for $x_1x_2x_3$ will lead to valid prefixes of 6-letter words.

With care, exercise 24's running time can be significantly decreased, once we realize that every node of the search tree for $1 \le l \le n$ requires testing 5757 possibilities for x_l in step B3. If we build a more elaborate data structure for the 5-letter words, so that it becomes easy to run though all words that have a specific letter in a specific position, we can refine the algorithm so that the average number of possibilities per level that need to be investigated becomes only

$$(5757.0,\ 1697.9,\ 844.1,\ 273.5,\ 153.5,\ 100.8); \qquad (14)$$

the total running time then drops to 1.15 teramems. Exercise 25 has the details. And exercise 28 discusses a method that's faster yet.

Commafree codes. Our next example deals entirely with *four*-letter words. But it's not obscene; it's an intriguing question of coding theory. The problem is to find a set of four-letter words that can be decoded even if we don't put spaces or other delimiters between them. If we take any message that's formed from words of the set by simply concatenating them together, `likethis`, and if we look at any seven consecutive letters $\ldots x_1x_2x_3x_4x_5x_6x_7 \ldots$, exactly one of the four-letter substrings $x_1x_2x_3x_4$, $x_2x_3x_4x_5$, $x_3x_4x_5x_6$, $x_4x_5x_6x_7$ will be a codeword. Equivalently, if $x_1x_2x_3x_4$ and $x_5x_6x_7x_8$ are codewords, then $x_2x_3x_4x_5$ and $x_3x_4x_5x_6$ and $x_4x_5x_6x_7$ aren't. (For example, `iket` isn't.) Such a set is called a "commafree code" or a "self-synchronizing block code" of length four.

Commafree codes were introduced by F. H. C. Crick, J. S. Griffith, and L. E. Orgel [*Proc. National Acad. Sci.* **43** (1957), 416–421], and studied further by S. W. Golomb, B. Gordon, and L. R. Welch [*Canadian Journal of Mathematics* **10** (1958), 202–209], who considered the general case of m-letter alphabets and n-letter words. They constructed optimum commafree codes for all m when $n = 2$, 3, 5, 7, 9, 11, 13, and 15; and optimum codes for all m were subsequently found also for $n = 17, 19, 21, \ldots$ (see exercise 32). We will focus our attention on the four-letter case here ($n = 4$), partly because that case is still very far from being resolved, but mostly because the task of finding such codes is especially instructive. Indeed, our discussion will lead us naturally to an understanding of several significant techniques that are important for backtrack programming in general.

teramems
profile
search tree
data structure
commafree codes–
4-letter codewords
four-letter words
coding theory
concatenating
codewords
self-synchronizing block code
block code
Crick
Griffith
Orgel
Golomb
Gordon
Welch

To begin, we can see immediately that a commafree codeword cannot be "periodic," like `dodo` or `gaga`. Such a word already appears within two adjacent copies of itself. Thus we're restricted to *aperiodic* words like `item`, of which there are $m^4 - m^2$. Notice further that if `item` has been chosen, we aren't allowed to include any of its cyclic shifts `temi`, `emit`, or `mite`, because they all appear within `itemitem`. Hence the maximum number of codewords in our commafree code cannot exceed $(m^4 - m^2)/4$.

periodic
aperiodic
cyclic shifts
prime strings
lookahead
dynamic ordering–
search rearrangement, see dynamic ordering
cutoff properties

For example, consider the binary case, $m = 2$, when this maximum is 3. Can we choose three four-bit "words," one from each of the cyclic classes

$$\begin{aligned}[0001] &= \{0001, 0010, 0100, 1000\},\\ [0011] &= \{0011, 0110, 1100, 1001\},\\ [0111] &= \{0111, 1110, 1101, 1011\},\end{aligned} \qquad (15)$$

so that the resulting code is commafree? Yes: One solution in this case is simply to choose the smallest word in each class, namely 0001, 0011, and 0111. (Alert readers will recall that we studied the smallest word in the cyclic class of *any* aperiodic string in Section 7.2.1.1, where such words were called *prime strings* and where some of the remarkable properties of prime strings were proved.)

That trick doesn't work when $m = 3$, however, when there are $(81 - 9)/4 = 18$ cyclic classes. Then we cannot include 1112 after we've chosen 0001 and 0011. Indeed, a code that contains 0001 and 1112 can't contain either 0011 or 0111.

We could systematically backtrack through 18 levels, choosing x_1 in [0001] and x_2 in [0011], etc., and rejecting each x_l as in Algorithm B whenever we discover that $\{x_1, x_2, \ldots, x_l\}$ isn't commafree. For example, if $x_1 = 0010$ and we try $x_2 = 1001$, this approach would backtrack because x_1 occurs inside x_2x_1.

But a naïve strategy of that kind, which recognizes failure only after a bad choice has been made, can be vastly improved. If we had been clever enough, we could have looked a little bit ahead, and never even considered the choice $x_2 = 1001$ in the first place. Indeed, after choosing $x_1 = 0010$, we can automatically exclude *all* further words of the form $*001$, such as 2001 when $m \ge 3$ and 3001 when $m \ge 4$.

Even better pruning occurs if, for example, we've chosen $x_1 = 0001$ and $x_2 = 0011$. Then we can immediately rule out all words of the forms $1{*}{*}{*}$ or ${*}{*}{*}0$, because $x_1 1{*}{*}{*}$ includes x_2 and ${*}{*}{*}0x_2$ includes x_1. Already we could then deduce, in the case $m \ge 3$, that classes [0002], [0021], [0111], [0211], and [1112] *must* be represented by 0002, 0021, 0111, 0211, and 2111, respectively; each of the other three possibilities in those classes has been wiped out!

Thus we see the desirability of a lookahead mechanism.

Dynamic ordering of choices. Furthermore, we can see from this example that it's not always good to choose x_1, then x_2, then x_3, and so on when trying to satisfy a general property $P_n(x_1, x_2, \ldots, x_n)$ in the setting of Algorithm B. Maybe the search tree will be much smaller if we first choose x_5, say, and then turn next to some other x_j, depending on the particular value of x_5 that was selected. Some orderings might have much better cutoff properties than others, and every branch of the tree is free to choose its variables in any desired order.

frame
move
sequential lists–
downdating vs updating
cache-friendly
unordered sequential list
cells of memory
MEM, an array of “cells”–

Indeed, our commafree coding problem for ternary 4-tuples doesn't dictate any particular ordering of the 18 classes that would be likely to keep the search tree small. Therefore, instead of calling those choices x_1, x_2, …, x_{18}, it's better to identify them by the various class names, namely x_{0001}, x_{0002}, x_{0011}, x_{0012}, x_{0021}, x_{0022}, x_{0102}, x_{0111}, x_{0112}, x_{0121}, x_{0122}, x_{0211}, x_{0212}, x_{0221}, x_{0222}, x_{1112}, x_{1122}, x_{1222}. (Algorithm 7.2.1.1F is a good way to generate those names.) At every node of the search tree we then can choose a convenient variable on which to branch, based on previous choices. After beginning with $x_{0001} \leftarrow 0001$ at level 1 we might decide to try $x_{0011} \leftarrow 0011$ at level 2; and then, as we've seen, the choices $x_{0002} \leftarrow 0002$, $x_{0021} \leftarrow 0021$, $x_{0111} \leftarrow 0111$, $x_{0211} \leftarrow 0211$, and $x_{1112} \leftarrow 2111$ are forced, so we should make them at levels 3 through 7.

Furthermore, after those forced moves are made, it turns out that they don't force any others. But only two choices for x_{0012} will remain, while x_{0122} will have three. Therefore it will probably be wiser to branch on x_{0012} rather than on x_{0122} at level 8. (Incidentally, it also turns out that there *is* no commafree code with $x_{0001} = 0001$ and $x_{0011} = 0011$, *except* when $m = 2$.)

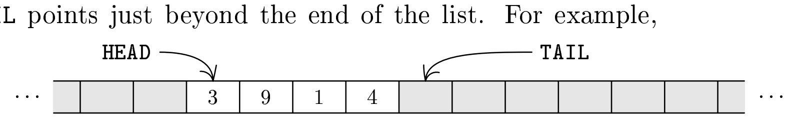
It's easy to adapt Algorithms B and W to allow dynamic ordering. Every node of the search tree can be given a “frame” in which we record the variable being set and the choice that was made. This choice of variable and value can be called a “move” made by the backtrack procedure.

Dynamic ordering can be helpful also after backtracking has taken place. If we continue the example above, where $x_{0001} = 0001$ and we've explored all cases in which $x_{0011} = 0011$, we aren't obliged to continue by trying another value for x_{0011}. We do want to remember that 0011 should no longer be considered legal, until x_{0001} changes; but we could decide to explore next a case such as $x_{0002} = 2000$ at level 2. In fact, $x_{0002} = 2000$ is quickly seen to be impossible in the presence of 0001 (see exercise 34). An even more efficient choice at level 2, however, is $x_{0012} = 0012$, because that branch immediately forces $x_{0002} = 0002$, $x_{0022} = 0022$, $x_{0122} = 0122$, $x_{0222} = 0222$, $x_{1222} = 1222$, and $x_{0011} = 1001$.

Sequential allocation redux. The choice of a variable and value on which to branch is a delicate tradeoff. We don't want to devote more time to planning than we'll save by having a good plan.

If we're going to benefit from dynamic ordering, we'll need efficient data structures that will lead to good decisions without much deliberation. On the other hand, elaborate data structures need to be updated whenever we branch to a new level, and they need to be downdated whenever we return from that level. Algorithm L illustrates an efficient mechanism based on linked lists; but sequentially allocated lists are often even more appealing, because they are cache-friendly and they involve fewer accesses to memory.

Assume then that we wish to represent a set of items as an unordered sequential list. The list begins in a cell of memory pointed to by HEAD, and TAIL points just beyond the end of the list. For example,



(16)

empty
insert
overflow
inverse list
inverse permutation
active

is one way to represent the set $\{1,3,4,9\}$. The number of items currently in the set is $\mathtt{TAIL}-\mathtt{HEAD}$; thus $\mathtt{TAIL}=\mathtt{HEAD}$ if and only if the list is empty. If we wish to insert a new item x, knowing that x isn't already present, we simply set

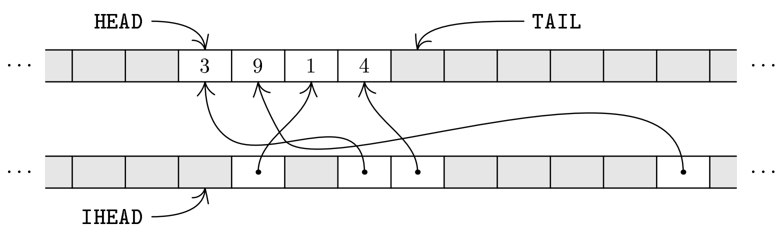
$$\mathtt{MEM[TAIL]} \leftarrow x,\quad \mathtt{TAIL}\leftarrow \mathtt{TAIL}+1. \qquad (17)$$

Conversely, if $\mathtt{HEAD}\le \mathtt{P} < \mathtt{TAIL}$, we can easily delete $\mathtt{MEM[P]}$:

$$\mathtt{TAIL}\leftarrow\mathtt{TAIL}-1;\quad \text{if } \mathtt{P}\ne\mathtt{TAIL}\text{, set } \mathtt{MEM[P]}\leftarrow\mathtt{MEM[TAIL]}. \qquad (18)$$

(We've tacitly assumed in (17) that MEM[TAIL] is available for use whenever a new item is inserted. Otherwise we would have had to test for memory overflow.)

We can't delete an item from a list without knowing its MEM location. Thus we will often want to maintain an "inverse list," assuming that all items x lie in the range $0\le x<M$. For example, (16) becomes the following, if $M=10$:



(19)

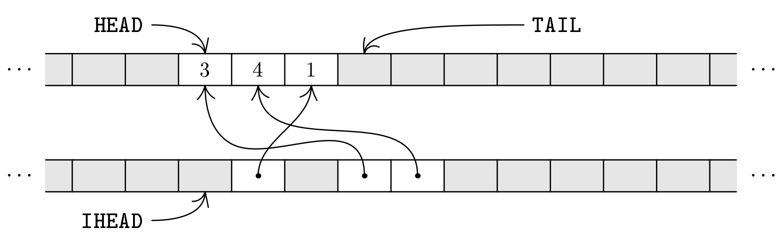
(Shaded cells have undefined contents.) With this setup, insertion (17) becomes

$$\mathtt{MEM[TAIL]}\leftarrow x,\quad \mathtt{MEM[IHEAD}+x\mathtt{]}\leftarrow\mathtt{TAIL},\quad \mathtt{TAIL}\leftarrow\mathtt{TAIL}+1, \qquad (20)$$

and TAIL will never exceed $\mathtt{HEAD}+M$. Similarly, deletion of x becomes

$$\mathtt{P}\leftarrow\mathtt{MEM[IHEAD}+x\mathtt{]},\quad \mathtt{TAIL}\leftarrow\mathtt{TAIL}-1;$$
$$\text{if } \mathtt{P}\ne\mathtt{TAIL}\text{, set } y\leftarrow\mathtt{MEM[TAIL]},\ \mathtt{MEM[P]}\leftarrow y,\ \mathtt{MEM[IHEAD}+y\mathtt{]}\leftarrow\mathtt{P}. \qquad (21)$$

For example, after deleting '9' from (19) we would obtain this:



(22)

In more elaborate situations we also want to test whether or not a given item x is present. If so, we can keep more information in the inverse list. A particularly useful variation arises when the list that begins at IHEAD contains a *complete* permutation of the values $\{\mathtt{HEAD},\mathtt{HEAD}+1,\ldots,\mathtt{HEAD}+M-1\}$, and the memory cells beginning at HEAD contain the inverse permutation — although only the first $\mathtt{TAIL}-\mathtt{HEAD}$ elements of that list are considered to be "active."

For example, in our commafree code problem with $m=3$, we can begin by putting items representing the $M=18$ cycle classes [0001], [0002], …, [1222] into memory cells HEAD through $\mathtt{HEAD}+17$. Initially they're all active, with

delete
data structures
periodic
radix m

TAIL = HEAD + 18 and MEM[IHEAD + c] = HEAD + c for $0 \le c < 18$. Then whenever we decide to choose a codeword for class c, we delete c from the active list by using a souped-up version of (21) that maintains full permutations:

$$\begin{aligned}&\mathtt{P} \leftarrow \mathtt{MEM[IHEAD}+c\mathtt{]},\quad \mathtt{TAIL} \leftarrow \mathtt{TAIL}-1;\\ &\quad \text{if } \mathtt{P} \ne \mathtt{TAIL}, \text{ set } y \leftarrow \mathtt{MEM[TAIL]},\quad \mathtt{MEM[TAIL]} \leftarrow c,\quad \mathtt{MEM[P]} \leftarrow y,\\ &\qquad\qquad\qquad \mathtt{MEM[IHEAD}+c\mathtt{]} \leftarrow \mathtt{TAIL},\quad \mathtt{MEM[IHEAD}+y\mathtt{]} \leftarrow \mathtt{P}.\end{aligned} \qquad (23)$$

Later on, after backtracking to a state where we once again want c to be considered active, we simply set TAIL ← TAIL + 1, because c will already be in place!

Lists for the commafree problem. The task of finding all four-letter commafree codes is not difficult when $m = 3$ and only 18 cycle classes are involved. But it already becomes challenging when $m = 4$, because we must then deal with $(4^4 - 4^2)/4 = 60$ classes. Therefore we'll want to give it some careful thought as we try to set it up for backtracking.

The example scenarios for $m = 3$ considered above suggest that we'll repeatedly want to know the answers to questions such as, "How many words of the form 02∗∗ are still available for selection as codewords?" Redundant data structures, oriented to queries of that kind, appear to be needed. Fortunately, we shall see that there's a nice way to provide them, using sequential lists as in (19)–(23).

In Algorithm C below, each of the m^4 four-letter words is given one of three possible states during the search for commafree codes. A word is *green* if it's part of the current set of tentative codewords. It is *red* if it's not currently a candidate for such status, either because it is incompatible with the existing green words or because the algorithm has already examined all scenarios in which it is green in their presence. Every other word is *blue*, and sort of in limbo; the algorithm might or might not decide to make it red or green. All words are initially blue — except for the m^2 periodic words, which are permanently red.

We'll use the Greek letter α to stand for the integer value of a four-letter word x in radix m. For example, if $m = 3$ and if x is the word 0102, then $\alpha = (0102)_3 = 11$. The current state of word x is kept in MEM[α], using one of the arbitrary internal codes 2 (GREEN), 0 (RED), or 1 (BLUE).

The most important feature of the algorithm is that *every blue word $x = x_1x_2x_3x_4$ is potentially present in seven different lists*, called P1(x), P2(x), P3(x), S1(x), S2(x), S3(x), and CL(x), where

- P1(x), P2(x), P3(x) are the blue words matching $x_1{*}{*}{*}$, $x_1x_2{*}{*}$, $x_1x_2x_3{*}$;
- S1(x), S2(x), S3(x) are the blue words matching ${*}{*}{*}x_4$, ${*}{*}x_3x_4$, ${*}x_2x_3x_4$;
- CL(x) hosts the blue words in $\{x_1x_2x_3x_4, x_2x_3x_4x_1, x_3x_4x_1x_2, x_4x_1x_2x_3\}$.

These seven lists begin respectively in MEM locations P1OFF+$p_1(\alpha)$, P2OFF+$p_2(\alpha)$, P3OFF+$p_3(\alpha)$, S1OFF+$s_1(\alpha)$, S2OFF+$s_2(\alpha)$, S3OFF+$s_3(\alpha)$, and CLOFF+$4cl(\alpha)$; here (P1OFF, P2OFF, P3OFF, S1OFF, S2OFF, S3OFF, CLOFF) are respectively ($2m^4$, $5m^4$, $8m^4$, $11m^4$, $14m^4$, $17m^4$, $20m^4$). We define $p_1((x_1x_2x_3x_4)_m) = (x_1)_m$, $p_2((x_1x_2x_3x_4)_m) = (x_1x_2)_m$, $p_3((x_1x_2x_3x_4)_m) = (x_1x_2x_3)_m$, $s_1((x_1x_2x_3x_4)_m) = (x_4)_m$, $s_2((x_1x_2x_3x_4)_m) = (x_3x_4)_m$, $s_3((x_1x_2x_3x_4)_m) = (x_2x_3x_4)_m$; and finally $cl((x_1x_2x_3x_4)_m)$ is an internal number between 0 and $(m^4-m^2)/4-1$ assigned

reflection
symmetry breaking
closed

Table 1

LISTS USED BY ALGORITHM C ($m = 2$), ENTERING LEVEL 1

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
0	RED	BLUE	BLUE	BLUE	RED	RED	BLUE	BLUE	RED	BLUE	RED	BLUE	BLUE	BLUE	BLUE	RED	
10		20	21	22			23	24		29		2c	28	2b	2a		
20	0001	0010	0011	0110	0111				1100	1001	1110	1101	1011				P1
30	25								2d								
40		50	51	52			54	55		58		59	5c	5e	5d		
50	0001	0010	0011		0110	0111			1001	1011			1100	1110	1101		P2
60	53				56				5a				5f				
70		80	82	83			86	87		88		8a	8c	8d	8e		
80	0001		0010	0011			0110	0111	1001		1011		1100	1101	1110		P3
90	81		84		84		88		89		8b		8e		8f		
a0		b8	b0	b9			b1	bb		ba		bd	b2	bc	b3		
b0	0010	0110	1100	1110					0001	0011	1001	0111	1101	1011			S1
c0	b4								be								
d0		e4	e8	ec			e9	ed		e5		ee	e0	e6	ea		
e0	1100				0001	1001	1101		0010	0110	1110		0011	0111	1011		S2
f0	e1				e7				eb				ef				
100		112	114	116			11c	11e		113		117	118	11a	11d		
110			0001	1001	0010		0011	1011	1100		1101		0110	1110	0111		S3
120	110		114		115		118		119		11b		11e		11f		
130		140	141	144			145	148		147		14b	146	14a	149		
140	0001	0010			0011	0110	1100	1001	0111	1110	1101	1011					CL
150	142				148				14c								

This table shows MEM locations 0000 through 150f, using hexadecimal notation. (For example, MEM[40d] = 5e; see exercise 36.) Blank entries are unused by the algorithm.

to each class. The seven MEM locations where x appears in these seven lists are respectively kept in inverse lists that begin in MEM locations $\texttt{P1OFF} - m^4 + \alpha$, $\texttt{P2OFF} - m^4 + \alpha$, ..., $\texttt{CLOFF} - m^4 + \alpha$. And the TAIL pointers, which indicate the current list sizes as in (19)–(23), are respectively kept in MEM locations $\texttt{P1OFF} + m^4 + \alpha$, $\texttt{P2OFF} + m^4 + \alpha$, ..., $\texttt{CLOFF} + m^4 + \alpha$. (Whew; got that?)

This vast apparatus, which occupies $22m^4$ cells of MEM, is illustrated in Table 1, at the beginning of the computation for the case $m = 2$. Fortunately it's not really as complicated as it may seem at first. Nor is it especially vast: After all, $22m^4$ is only 13,750 when $m = 5$.

(A close inspection of Table 1 reveals incidentally that the words 0100 and 1000 have been colored red, not blue. That's because we can assume without loss of generality that class [0001] is represented either by 0001 or by 0010. The other two cases are covered by left-right reflection of all codewords.)

Algorithm C finds these lists invaluable when it is deciding where next to branch. But it has no further use for a list in which one of the items has become green. Therefore it declares such lists "closed"; and it saves most of the work of list maintenance by updating *only* the lists that remain open. A closed list is represented internally by setting its TAIL pointer to $\texttt{HEAD} - 1$.

For example, Table 2 shows how the lists in MEM will have changed just after $x = 0010$ has been chosen to be a tentative codeword. The elements $\{0001, 0010, 0011, 0110, 0111\}$ of $\texttt{P1}(x)$ are effectively hidden, because the tail pointer $\texttt{MEM[30]} = \texttt{1f} = \texttt{20} - 1$ marks that list as closed. (Those list elements ac-

undoing–
Floyd
compiler

Table 2

LISTS USED BY ALGORITHM C ($m = 2$), ENTERING LEVEL 2

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	
0	RED	RED	GREEN	BLUE	RED	RED	BLUE	BLUE	RED	RED	RED	BLUE	BLUE	BLUE	BLUE	RED	
10												29	28	2b	2a		
20									1100	1011	1110	1101					P1
30	1f								2c								
40							54	55				58	5c	5e	5d		
50					0110	0111			1011				1100	1110	1101		P2
60	4f				56				59				5f				
70							86	87				8a	8c	8d	8e		
80							0110	0111			1011		1100	1101	1110		P3
90	80		81		84		88		88		8b		8e		8f		
a0				b9				bb				b8		ba			
b0									1011	0011	1101	0111					S1
c0	af								bc								
d0				ec				ed				ee	e0	e4			
e0	1100				1101								0011	0111	1011		S2
f0	e1				e5				e7				ef				
100				116			11c	11e				117	118	11a	11d		
110							0011	1011	1100		1101		0110	1110	0111		S3
120	110		112		113		118		119		11b		11e		11f		
130				144			145	148				14b	146	14a	149		
140					0011	0110	1100		0111	1110	1101	1011					CL
150	13f				147				14c								

The word 0010 has become green, thus closing its seven lists and making 0001 red. The logic of Algorithm C has also made 1001 red. Hence 0001 and 1001 have been deleted from the open lists in which they formerly appeared (see exercise 37).

tually do still appear in MEM locations 200 through 204, just as they did in Table 1. But there's no need to look at that list while any word of the form 0$***$ is green.)

A general mechanism for doing and undoing. We're almost ready to finalize the details of Algorithm C and to get on with the search for commafree codes, but a big problem still remains: The state of computation at every level of the search involves all of the marvelous lists that we've just specified, and those lists aren't tiny. They occupy more than 5000 cells of MEM when $m = 4$, and they can change substantially from level to level.

We could make a new copy of the entire state, whenever we advance to a new node of the search tree. But that's a bad idea, because we don't want to perform thousands of memory accesses per node. A much better strategy would be to stick with a single instance of MEM, and to update and downdate the lists as the search progresses, if we could only think of a simple way to do that.

And we're in luck: There *is* such a way, first formulated by R. W. Floyd in his classic paper "Nondeterministic algorithms" [*JACM* **14** (1967), 636–644]. Floyd's original idea, which required a special compiler to generate forward and backward versions of every program step, can in fact be greatly simplified when all of the changes in state are confined to a single MEM array. All we need to do is to replace every assignment operation of the form '$\texttt{MEM}[a] \leftarrow v$' by the

UNDO
stack
reversible-memory
stamping
fallback point
bump
overflow
lookahead
poison list

slightly more cumbersome operation

$$\text{store}(a,v): \;\; \text{Set } \texttt{UNDO[}u\texttt{]} \leftarrow \big(a, \texttt{MEM[}a\texttt{]}\big),\;\; \texttt{MEM[}a\texttt{]} \leftarrow v, \text{ and } u \leftarrow u+1. \qquad (24)$$

Here UNDO is a sequential stack that holds (address, value) pairs; in our application we could say '$\texttt{UNDO[}u\texttt{]} \leftarrow (a \ll 16) + \texttt{MEM[}a\texttt{]}$', because the cell addresses and values never exceed 16 bits. Of course we'll also need to check that the stack pointer u doesn't get too large, if the number of assignments has no a priori limit.

Later on, when we want to undo all changes to MEM since the time when u had reached a particular value u_0, we simply do this:

$$\text{unstore}(u_0): \;\; \text{While } u > u_0,\; \text{ set } u \leftarrow u-1, \\ (a,v) \leftarrow \texttt{UNDO[}u\texttt{]},\;\text{ and } \texttt{MEM[}a\texttt{]} \leftarrow v. \qquad (25)$$

In our application the unstacking operation '$(a,v) \leftarrow \texttt{UNDO[}u\texttt{]}$' here could be implemented by saying '$a \leftarrow \texttt{UNDO[}u\texttt{]} \gg 16,\;\; v \leftarrow \texttt{UNDO[}u\texttt{]} \,\&\, \texttt{\#ffff}$'.

A useful refinement of this reversible-memory technique is often advantageous, based on the idea of "stamping" that is part of the folklore of programming. It puts only one item on the UNDO stack when the same memory address is updated more than once in the same round.

$$\text{store}(a,v): \;\; \text{If } \texttt{STAMP[}a\texttt{]} \neq \sigma, \text{ set } \texttt{STAMP[}a\texttt{]} \leftarrow \sigma, \\ \texttt{UNDO[}u\texttt{]} \leftarrow \big(a, \texttt{MEM[}a\texttt{]}\big), \text{ and } u \leftarrow u+1. \\ \text{Then set } \texttt{MEM[}a\texttt{]} \leftarrow v. \qquad (26)$$

Here STAMP is an array with one entry for each address in MEM. It's initially all zero, and σ is initially 1. Whenever we come to a fallback point, where the current stack pointer will be remembered as the value u_0 for some future undoing, we "bump" the current stamp by setting $\sigma \leftarrow \sigma + 1$. Then (26) will continue to do the right thing. (In programs that run for a long time, we must be careful when integer overflow causes σ to be bumped to zero; see exercise 38.)

Notice that the combination of (24) and (25) will perform five memory accesses for each assignment and its undoing. The combination of (26) and (25) will cost seven mems for the first assignment to $\texttt{MEM[}a\texttt{]}$, but only two mems for every subsequent assignment to the same address. So (26) wins, if multiple assignments exceed one-time-only assignments.

Backtracking through commafree codes. OK, we're now equipped with enough basic knowhow to write a pretty good backtrack program for the problem of generating all commafree four-letter codes.

Algorithm C below incorporates one more key idea, which is a lookahead mechanism that is specific to commafree backtracking; we'll call it the "poison list." Every item on the poison list is a pair, consisting of a suffix and a prefix that the commafree rule forbids from occurring together. Every green word $x_1x_2x_3x_4$ — that is, every word that will be a final codeword in the current branch of our backtrack search — contributes three items to the poison list, namely

$$(*x_1x_2x_3, x_4{*}{*}{*}), \quad ({*}{*}x_1x_2, x_3x_4{*}{*}), \quad \text{and} \quad ({*}{*}{*}x_1, x_2x_3x_4*). \qquad (27)$$

If there's a green word on both sides of a poison list entry, we're dead: The commafree condition fails, and we must backtrack. If there's a green word on one side but not the other, we can kill off all blue words on the other side by making them red. And if either side of a poison list entry corresponds to an *empty* list, we can remove this entry from the poison list because it will never affect the outcome. (Blue words become red or green, but red words stay red.)

inverse list
Laxdal
Jiggs
stamping++

For example, consider the transition from Table 1 to Table 2. When word 0010 becomes green, the poison list receives its first three items:

$$(*001, 0{*}{*}{*}), \quad ({*}{*}00, 10{*}{*}), \quad ({*}{*}{*}0, 010*).$$

The first of these kills off the $*001$ list, because $0{*}{*}{*}$ contains the green word 0010. That makes 1001 red. The last of these, similarly, kills off the $010*$ list; but that list is empty when $m = 2$. The poison list now reduces to a single item, $({*}{*}00, 10{*}{*})$, which remains poisonous because list $**00$ contains the blue word 1100 and $10{*}{*}$ contains the blue word 1011.

We'll maintain the poison list at the end of MEM, following the CL lists. It obviously will contain at most $3(m^4 - m^2)/4$ entries, and in fact it usually turns out to be quite small. No inverse list is required; so we shall adopt the simple method of (17) and (18), but with two cells per entry so that TAIL will change by ± 2 instead of by ± 1. The value of TAIL will be stored in MEM at key times so that temporary changes to it can be undone.

The case $m = 4$, in which each codeword consists of four *quaternary* digits $\{0, 1, 2, 3\}$, is particularly interesting, because an early backtrack program by Lee Laxdal found that no such commafree code can make use of all 60 of the cycle classes [0001], [0002], ..., [2333]. [See B. H. Jiggs, *Canadian Journal of Math.* **15** (1963), 178–187.] Laxdal's program also reportedly showed that at least three of those classes must be omitted; and it found several valid 57-word sets. Further details were never published, because the proof that 58 codewords are impossible depended on what Jiggs called a "quite time-consuming" computation.

Because size 60 is impossible, our algorithm cannot simply assume that a move such as 1001 is forced when the other words 0011, 0110, 1100 of its class have been ruled out. We must also consider the possibility that class [0011] is entirely absent from the code. Such considerations add an interesting further twist to the problem, and Algorithm C describes one way to cope with it.

Algorithm C (*Four-letter commafree codes*). Given an alphabet size $m \le 7$ and a goal g in the range $L - m(m-1) \le g \le L$, where $L = (m^4 - m^2)/4$, this algorithm finds all sets of g four-letter words that are commafree and include either 0001 or 0010. It uses an array MEM of $M = \lfloor 23.5m^4 \rfloor$ 16-bit numbers, as well as several more auxiliary arrays: ALF of size $16^3 m$; STAMP of size M; X, C, S, and U of size $L + 1$; FREE and IFREE of size L; and a sufficiently large array called UNDO whose maximum size is difficult to guess.

C1. [Initialize.] Set ALF[$(abcd)_{16}$] $\leftarrow (abcd)_m$ for $0 \le a, b, c, d < m$. Set STAMP[k] $\leftarrow 0$ for $0 \le k < M$ and $\sigma \leftarrow 0$. Put the initial prefix, suffix, and class lists into MEM, as in Table 1. Also create an empty poison list by

setting MEM[PP] $\leftarrow$ POISON, where POISON $= 22m^4$ and PP $=$ POISON $- 1$. Set FREE[k] $\leftarrow$ IFREE[k] $\leftarrow k$ for $0 \le k < L$. Then set $l \leftarrow 1$, $x \leftarrow$ #0001, $c \leftarrow 0$, $s \leftarrow L - g$, $f \leftarrow L$, $u \leftarrow 0$, and go to step C3. (Variable l is the level, x is a trial word, c is its class, s is the "slack," f is the number of free classes, and u is the size of the UNDO stack.)

C2. [Enter level l.] If $l > L$, visit the solution $x_1 \ldots x_L$ and go to C6. Otherwise choose a candidate word x and class c as described in exercise 39.

C3. [Try the candidate.] Set U[l] $\leftarrow u$ and $\sigma \leftarrow \sigma + 1$. If $x < 0$, go to C6 if $s = 0$ or $l = 1$, otherwise set $s \leftarrow s - 1$. If $x \ge 0$, update the data structures to make x green, as described in exercise 40, escaping to C5 if trouble arises.

C4. [Make the move.] Set X[l] $\leftarrow x$, C[l] $\leftarrow c$, S[l] $\leftarrow s$, $p \leftarrow$ IFREE[c], $f \leftarrow f - 1$. If $p \ne f$, set $y \leftarrow$ FREE[f], FREE[p] $\leftarrow y$, IFREE[y] $\leftarrow p$, FREE[f] $\leftarrow c$, IFREE[c] $\leftarrow f$. (This is (23).) Then set $l \leftarrow l + 1$ and go to C2.

C5. [Try again.] While $u >$ U[l], set $u \leftarrow u - 1$ and MEM[UNDO[u] $\gg$ 16] $\leftarrow$ UNDO[u] & #ffff. (Those operations restore the previous state, as in (25).) Then $\sigma \leftarrow \sigma + 1$ and redden x (see exercise 40). Go to C2.

C6. [Backtrack.] Set $l \leftarrow l - 1$, and terminate if $l = 0$. Otherwise set $x \leftarrow$ X[l], $c \leftarrow$ C[l], $f \leftarrow f - 1$. If $x < 0$, repeat this step (class c was omitted from the code). Otherwise set $s \leftarrow$ S[l] and go back to C5. ▮

Exercises 39 and 40 provide the instructive details that flesh out this skeleton.

Algorithm C needs just 13, 177, and 2380 megamems to prove that no solutions exist for $m = 4$ when g is 60, 59, and 58. It needs about 22800 megamems to find the 1152 solutions for $g = 57$; see exercise 44. There are roughly (14, 240, 3700, 38000) thousand nodes in the respective search trees, with most of the activity taking place on levels 30 ± 10. The height of the UNDO stack never exceeds 2804, and the poison list never contains more than 12 entries at a time.

Running time estimates. Backtrack programs are full of surprises. Sometimes they produce instant answers to a supposedly difficult problem. But sometimes they spin their wheels endlessly, trying to traverse an astronomically large search tree. And sometimes they deliver results just about as fast as we might expect.

Fortunately, we needn't sit in the dark. There's a simple Monte Carlo algorithm by which we can often tell in advance whether or not a given backtrack strategy will be feasible. This method, based on random sampling, can actually be worked out by hand *before* writing a program, in order to help decide whether to invest further time while following a particular approach. In fact, the very act of carrying out this pleasant pencil-and-paper method often suggests useful cutoff strategies and/or data structures that will be valuable later when a program is being written. For example, the author developed Algorithm C above after first doing some armchair experiments with random choices of potential commafree codewords, and noticing that a family of lists such as those in Tables 1 and 2 would be quite helpful when making further choices.

To illustrate the method, let's consider the n queens problem again, as represented in Algorithm B* above. When $n = 8$, we can obtain a decent "ballpark

Running time
estimates of run time–
Monte Carlo algorithm
random sampling
pencil-and-paper method
cutoff strategies
data structures
author

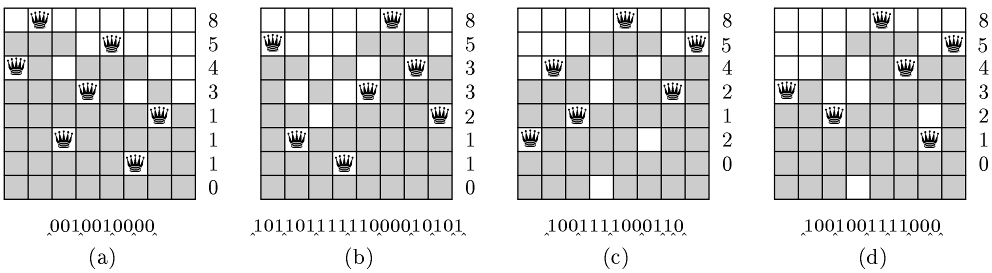


Fig. 69. Four random attempts to solve the 8 queens problem. Such experiments help to estimate the size of the backtrack tree in Fig. 68. The branching degrees are shown at the right of each diagram, while the random bits used for sampling appear below. Cells have been shaded in gray if they are attacked by one or more queens in earlier rows.

search tree
root node
degree
pi as source+
rejection method
e, as source
phi, as source
gamma, as source
random variables
cost function

estimate" of the size of Fig. 68 by examining only a few random paths in that search tree. We start by writing down the number $D_1 \leftarrow 8$, because there are eight ways to place the queen in row 1. (In other words, the root node of the search tree has degree 8.) Then we use a source of random numbers — say the binary digits of $\pi \bmod 1 = (.001001000011\ldots)_2$ — to select one of those placements. Eight choices are possible, so we look at three of those bits; we shall set $X_1 \leftarrow 2$, because 001 is the second of the eight possibilities (000, 001, ..., 111).

Given $X_1 = 2$, the queen in row 2 can't go into columns 1, 2, or 3. Hence five possibilities remain for X_2, and we write down $D_2 \leftarrow 5$. The next three bits of π lead us to set $X_2 \leftarrow 5$, since 5 is the second of the available columns (4, 5, 6, 7, 8) and 001 is the second value of (000, 001, ..., 100). If π had continued with 101 or 110 or 111 instead of 001, we would incidentally have used the "rejection method" of Section 3.4.1 and moved to the next three bits; see exercise 47.

Continuing in this way leads to $D_3 \leftarrow 4$, $X_3 \leftarrow 1$; then $D_4 \leftarrow 3$, $X_4 \leftarrow 4$. (Here we used the two bits 00 to select X_3, and the next two bits 00 to select X_4.) The remaining branches are forced: $D_5 \leftarrow 1$, $X_5 \leftarrow 7$; $D_6 \leftarrow 1$, $X_6 \leftarrow 3$; $D_7 \leftarrow 1$, $X_7 \leftarrow 6$; and we're stuck when we reach level 8 and find $D_8 \leftarrow 0$.

These sequential random choices are depicted in Fig. 69(a), where we've used them to place each queen successively into an unshaded cell. Parts (b), (c), and (d) of Fig. 69 correspond in the same way to choices based on the binary digits of $e \bmod 1$, $\phi \bmod 1$, and $\gamma \bmod 1$. Exactly 10 bits of π, 20 bits of e, 13 bits of ϕ, and 13 bits of γ were used to generate these examples.

In this discussion the notation D_k stands for a branching degree, not for a domain of values. We've used uppercase letters for the numbers D_1, X_1, D_2, etc., because those quantities are random variables. Once we've reached $D_l = 0$ at some level, we're ready to estimate the overall cost, by implicitly assuming that the path we've taken is representative of *all* root-to-leaf paths in the tree.

The cost of a backtrack program can be assessed by summing the individual amounts of time spent at each node of the search tree. Notice that every node on level l of that tree can be labeled uniquely by a sequence $x_1 \ldots x_{l-1}$, which defines the path from the root to that node. Thus our goal is to estimate the sum of all $c(x_1 \ldots x_{l-1})$, where $c(x_1 \ldots x_{l-1})$ is the cost associated with node $x_1 \ldots x_{l-1}$.

For example, the four queens problem is represented by the search tree (4), and its cost is the sum of 17 individual costs

subtree
size of the tree
standard deviation

$$c() + c(1) + c(13) + c(14) + c(142) + c(2) + c(24) + \cdots + c(413) + c(42). \qquad (28)$$

If $C(x_1 \ldots x_l)$ denotes the total cost of the subtree rooted at $x_1 \ldots x_l$, then

$$C(x_1 \ldots x_l) = c(x_1 \ldots x_l) + C(x_1 \ldots x_l x_{l+1}^{(1)}) + \cdots + C(x_1 \ldots x_l x_{l+1}^{(d)}) \qquad (29)$$

when the choices for x_{l+1} at node $x_1 \ldots x_l$ are $\{x_{l+1}^{(1)}, \ldots, x_{l+1}^{(d)}\}$. For instance in (4) we have $C(1) = c(1) + C(13) + C(14)$; $C(13) = c(13)$; and $C() = c() + C(1) + C(2) + C(3) + C(4)$ is the overall cost (28).

In these terms a Monte Carlo estimate for $C()$ is extremely easy to compute:

Theorem E. *Given D_1, X_1, D_2, X_2, … as above, the cost of backtracking is*

$$C() = \mathrm{E}\bigl(c() + D_1(c(X_1) + D_2(c(X_1X_2) + D_3(c(X_1X_2X_3) + \cdots)))\bigr). \qquad (30)$$

Proof. Node $x_1 \ldots x_l$, with branch degrees $d_1, \ldots, d_l$ above it, is reached with probability $1/d_1 \ldots d_l$; so it contributes $d_1 \ldots d_l c(x_1 \ldots x_l)/d_1 \ldots d_l = c(x_1 \ldots x_l)$ to the expected value in this formula. ▮

For example, the tree (4) has six root-to-leaf paths, and they occur with respective probabilities 1/8, 1/8, 1/4, 1/4, 1/8, 1/8. The first one contributes 1/8 times $c() + 4(c(1) + 2(c(13)))$, namely $c()/8 + c(1)/2 + c(13)$, to the expected value. The second contributes $c()/8 + c(1)/2 + c(14) + c(142)$; and so on.

A special case of Theorem E, with all $c(x_1 \ldots x_l) = 1$, tells us how to estimate the total size of the tree, which is often a crucial quantity:

Corollary E. *The number of nodes in the search tree, given D_1, D_2, …, is*

$$\mathrm{E}(1 + D_1 + D_1D_2 + \cdots) = \mathrm{E}\bigl(1 + D_1\bigl(1 + D_2(1 + D_3(1 + \cdots))\bigr)\bigr). \qquad (31)$$

For example, Fig. 69 gives us four estimates for the size of the tree in Fig. 68, using the numbers D_j at the right of each 8×8 diagram. The estimate from Fig. 69(a) is $1 + 8\bigl(1 + 5\bigl(1 + 4(1 + 3(1 + 1(1 + 1(1 + 1))))\bigr)\bigr) = 2129$; and the other three are respectively 2689, 1489, 2609. None of them is extremely far from the true number, 2057, although we can't expect to be so lucky all the time.

The detailed study in exercise 51 shows that the estimate (31) in the case of 8 queens turns out to be quite well behaved:

$$\bigl(\text{min } 489, \quad \text{ave } 2057, \quad \text{max } 7409, \quad \text{dev } \sqrt{1146640} \approx 1071\bigr). \qquad (32)$$

The analogous problem for 16 queens has a much less homogeneous search tree:

$$\bigl(\text{min } 2597105, \ \text{ave } 1141190303, \ \text{max } 131048318769, \ \text{dev} \approx 1234000000\bigr). \qquad (33)$$

Still, this standard deviation is roughly the same as the mean, so we'll usually guess the correct order of magnitude. (For example, ten independent experiments predicted .632, .866, .237, 1.027, 4.006, .982, .143, .140, 3.402, and .510 billion nodes, respectively. The mean of these is 1.195.) A thousand trials with $n = 64$ suggest that the problem of 64 queens will have about 3×10^{65} nodes in its tree.

binomial tree
estimating solutions

Let's formulate this estimation procedure precisely, so that it can be performed conveniently by machine as well as by hand:

Algorithm E (*Estimated cost of backtrack*). Given domains D_k and properties P_l as in Algorithm B, together with node costs $c(x_1 \ldots x_l)$ as above, this algorithm computes the quantity S whose expected value is the total cost $C()$ in (30). It uses an auxiliary array $y_1 y_2 \ldots$ whose size should be $\geq \max(|D_1|, \ldots, |D_n|)$.

E1. [Initialize.] Set $l \leftarrow D \leftarrow 1$, $S \leftarrow 0$, and initialize any data structures needed.

E2. [Enter level l.] (At this point $P_{l-1}(X_1, \ldots, X_{l-1})$ holds.) Set $S \leftarrow S + D \cdot c(X_1 \ldots X_{l-1})$. If $l > n$, terminate the algorithm. Otherwise set $d \leftarrow 0$ and set $x \leftarrow \min D_l$, the smallest element of D_l.

E3. [Test x.] If $P_l(X_1, \ldots, X_{l-1}, x)$ holds, set $y_d \leftarrow x$ and $d \leftarrow d + 1$.

E4. [Try again.] If $x \neq \max D_l$, set x to the next larger element of D_l and return to step E3.

E5. [Choose and try.] If $d = 0$, terminate. Otherwise set $D \leftarrow D \cdot d$ and $X_l \leftarrow y_I$, where I is a uniformly random integer in $\{0, \ldots, d-1\}$. Update the data structures to facilitate testing P_{l+1}, set $l \leftarrow l + 1$, and go back to E2. ▮

Although Algorithm E looks rather like Algorithm B, it never backtracks.

Of course we can't expect this algorithm to give decent estimates in cases where the backtrack tree is wildly erratic. The *expected* value of S, namely $\mathrm{E}\,S$, is indeed the true cost; but the *probable* values of S might be quite different.

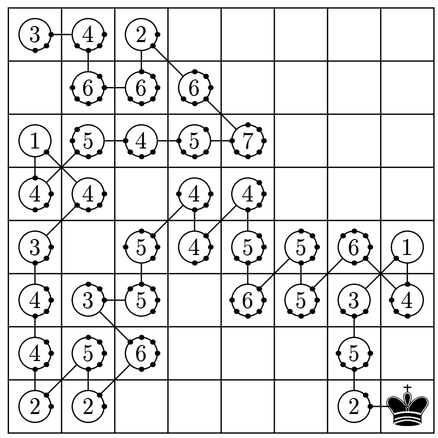
An extreme example of bad behavior occurs if property P_l is the simple condition '$x_1 > \cdots > x_l$' and all domains are $\{1, \ldots, n\}$. Then there's only one solution, $x_1 \ldots x_n = n \ldots 1$; and backtracking is a particularly stupid way to find it!

The search tree for this somewhat ridiculous problem is, nevertheless, quite interesting. It is none other than the binomial tree T_n of Eq. 7.2.1.3–(21), which has $\binom{n}{l}$ nodes on level $l + 1$ and 2^n nodes in total. If we set all costs to 1, the expected value of S is therefore $2^n = e^{n \ln 2}$. But exercise 50 proves that S will almost always be much smaller, less than $e^{(\ln n)^2 \ln \ln n}$. Furthermore the average value of l when Algorithm E terminates with respect to T_n is only $H_n + 1$. When $n = 100$, for example, the probability that $l \geq 20$ on termination is only 0.0000000027, while the vast majority of the nodes are near level 51.

Many refinements of Algorithm E are possible. For example, exercise 52 shows that the choices in step E5 need not be uniform. We shall discuss improved estimation techniques in Section 7.2.2.9, after having seen numerous examples of backtracking in practice.

***Estimating the number of solutions.** Sometimes we know that a problem has more solutions than we could ever hope to generate, yet we still want to know roughly how many there are. Algorithm E will tell us the approximate number, in cases where the backtrack process never reaches a dead end — that is, if it never terminates with $d = 0$ in step E5. There may be another criterion for successful termination in step E2 even though l might still be $\leq n$. The expected final value of D is exactly the total number of solutions, because every solution $X_1 \ldots X_l$ constructed by the algorithm is obtained with probability $1/D$.

For example, suppose we want to know the number of different paths by which a king can go from one corner of a chessboard to the opposite corner, without revisiting any square. One such path, chosen at random using the bits of π for guidance as we did in Fig. 69(a), is shown here. Starting in the upper left corner, we have 3 choices for the first move. Then, after moving to the right, there are 4 choices for the second move. And so on. We never make a move that would disconnect us from the goal; in particular, two of the moves are actually forced. (Exercise 58 explains one way to avoid fatal mistakes.)



The probability of obtaining this particular path is exactly $\frac{1}{3}\frac{1}{4}\frac{1}{6}\frac{1}{6}\frac{1}{2}\frac{1}{6}\frac{1}{7}\cdots\frac{1}{2} = 1/D$, where $D = 3\times4\times6\times6\times2\times6\times7\times\cdots\times2 = 1^2\cdot2^4\cdot3^4\cdot4^{10}\cdot5^9\cdot6^6\cdot7^1 \approx 8.7\times10^{20}$. Thus we can reasonably guess, at least tentatively, that there are 10^{21} such paths, more or less.

Of course that guess, based on a single random sample, rests on very shaky grounds. But we know that the average value $M_N = (D^{(1)} + \cdots + D^{(N)})/N$ of N guesses, in N independent experiments, will almost surely approach the correct number.

How large should N be, before we can have any confidence in the results? The actual values of D obtained from random king paths tend to vary all over the map. Figure 70 plots typical results, as N varies from 1 to 10000. For each value of N we can follow the advice of statistics textbooks and calculate the sample variance $V_N = S_N/(N-1)$ as in Eq. 4.2.2–(16); then $M_N \pm \sqrt{V_N/N}$ is the textbook estimate. The top diagram in Fig. 70 shows these "error bars" in gray, surrounding black dots for M_N. This sequence M_N does appear to settle down after N reaches 3000 or so, and to approach a value near 5×10^{25}. That's much higher than our first guess, but it has lots of evidence to back it up.

On the other hand, the bottom chart in Fig. 70 shows the distribution of the *logarithms* of the 10000 values of D that were used to make the top chart. Almost half of those values were totally negligible — less than 10^{20}. About 75% of them were less than 10^{24}. But some of them* exceeded 10^{28}. Can we really rely on a result that's based on such chaotic behavior? Is it really right to throw away most of our data and to trust almost entirely on observations that were obtained from comparatively few rare events?

Yes, we're okay! Some of the justification appears in exercise MPR–124, which is based on theoretical work by P. Diaconis and S. Chatterjee. In the paper cited with that exercise, they defend a simple measure of quality,

$$Q_N = \max(D^{(1)},\ldots,D^{(N)})/(NM_N) = \frac{\max(D^{(1)},\ldots,D^{(N)})}{D^{(1)}+\cdots+D^{(N)}}, \qquad (34)$$

king
chessboard
simple paths
paths, simple
π for guidance
statistics
sample variance
variance
error bars
discarded data
Diaconis
Chatterjee

* Four of the actual values that led to Fig. 70 were larger than 10^{28}; the largest, $\approx 2.1\times10^{28}$, came from a path of length 57. The smallest estimate, 19361664, came from a path of length 10.

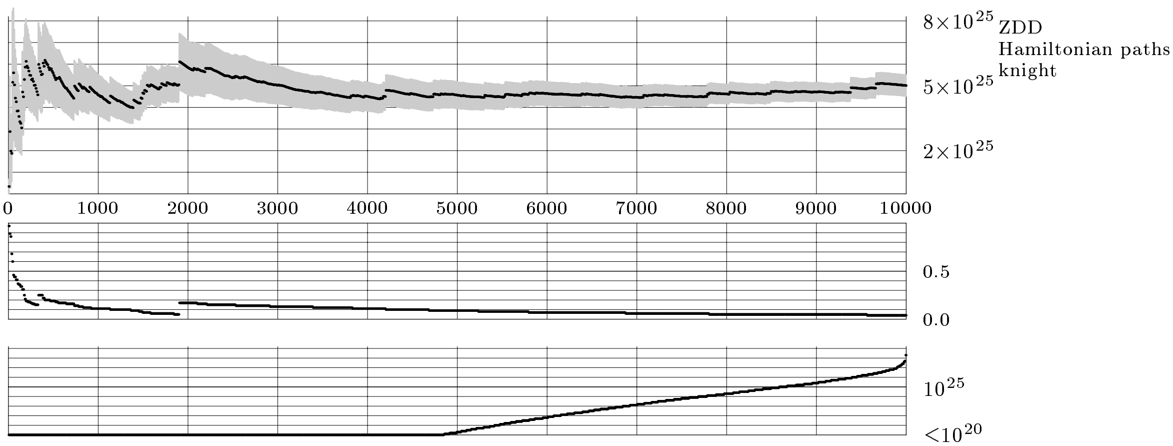


Fig. 70. Estimates of the number of king paths, based on up to 10000 random trials. The middle graph shows the corresponding quality measures of Eq. (34). The lower graph shows the *logarithms* of the individual estimates $D^{(k)}$, after they've been sorted.

arguing that a reasonable policy in most experiments such as these is to stop sampling when Q_N gets small. (Values of this statistic Q_N have been plotted in the middle of Fig. 70.)

Furthermore we can estimate other properties of the solutions to a backtrack problem, instead of merely counting those solutions. For example, the expected value of lD on termination of the random king's path algorithm is the total *length* of such paths. The data underlying Fig. 70 suggests that this total is $(2.66 \pm .14) \times 10^{27}$; hence the average path length appears to be about 53. The samples also indicate that about 34% of the paths pass through the center; about 46% touch the upper right corner; about 22% touch both corners; and about 7% pass through the center and both corners.

For this particular problem we don't actually need to rely on estimates, because the ZDD technology of Section 7.1.4 allows us to compute the *true* values. (See exercise 59.) The total number of simple corner-to-corner king paths on a chessboard is exactly 50,819,542,770,311,581,606,906,543; this value lies almost within the error bars of Fig. 70 for all $N \geq 250$, except for a brief interval near $N = 1400$. And the total length of all these paths turns out to be exactly 2,700,911,171,651,251,701,712,099,831, which is a little higher than our estimate. The true average length is therefore ≈ 53.15. The true probabilities of hitting the center, a given corner, both corners, and all three of those spots are respectively about 38.96%, 50.32%, 25.32%, and 9.86%.

The total number of corner-to-corner king paths of the maximum length, 63, is 2,811,002,302,704,446,996,926. This is a number that can *not* be estimated well by a method such as Algorithm E without additional heuristics.

The analogous problem for corner-to-corner *knight* paths, of any length, lies a bit beyond ZDD technology because many more ZDD nodes are needed. Using Algorithm E we can estimate that there are about $(8.6 \pm 1.2) \times 10^{19}$ such paths.

Historical notes. The origins of backtrack programming are obscure. Equivalent ideas must have occurred to many people, yet there was hardly any reason to write them down until computers existed. We can be reasonably sure that James Bernoulli used such principles in the 17th century, when he successfully solved the "Tot tibi sunt dotes" problem that had eluded so many others (see Section 7.2.1.7), because traces of the method exist in his exhaustive list of solutions.

Backtrack programs typically traverse the tree of possibilities by using what is now called depth-first search, a general graph exploration procedure that Édouard Lucas credited to a student named Trémaux [*Récréations Mathématiques* **1** (Paris: Gauthier-Villars, 1882), 47–50].

The eight queens problem was first proposed by Max Bezzel [*Schachzeitung* **3** (1848), 363; **4** (1849), 40] and by Franz Nauck [*Illustrirte Zeitung* **14**, 361 (1 June 1850), 352; **15**, 377 (21 September 1850), 182], perhaps independently. C. F. Gauss saw the latter publication, and wrote several letters about it to his friend H. C. Schumacher. Gauss's letter of 27 September 1850 is especially interesting, because it explained how to find all the solutions by backtracking— which he called 'Tatonniren', from a French term meaning "to feel one's way." He also listed the lexicographically first solutions of each equivalence class under reflection and rotation: 15863724, 16837425, 24683175, 25713864, 25741863, 26174835, 26831475, 27368514, 27581463, 35281746, 35841726, and 36258174.

Computers arrived a hundred years later, and people began to use them for combinatorial problems. The time was therefore ripe for backtracking to be described as a general technique, and Robert J. Walker rose to the occasion [*Proc. Symposia in Applied Math.* **10** (1960), 91–94]. His brief note introduced Algorithm W in machine-oriented form, and mentioned that the procedure could readily be extended to find variable-length patterns $x_1 \ldots x_n$ where n is not fixed.

The next milestone was a paper by Solomon W. Golomb and Leonard D. Baumert [*JACM* **12** (1965), 516–524], who formulated the general problem carefully and presented a variety of examples. In particular, they discussed the search for maximum commafree codes, and noted that backtracking can be used to find successively better and better solutions to combinatorial optimization problems. They introduced certain kinds of lookahead, as well as the important idea of dynamic ordering by branching on variables with the fewest remaining choices.

Other noteworthy early discussions of backtrack programming appear in Mark Wells's book *Elements of Combinatorial Computing* (1971), Chapter 4; in a survey by J. R. Bitner and E. M. Reingold, *CACM* **18** (1975), 651–656; and in the Ph.D. thesis of John Gaschnig [Report CMU-CS-79-124 (Carnegie Mellon University, 1979), Chapter 4]. Gaschnig introduced techniques of "backmarking" and "backjumping" that we shall discuss later.

Monte Carlo estimates of the cost of backtracking were first described briefly by M. Hall, Jr., and D. E. Knuth in *Computers and Computing, AMM* **72**,2, part 2, Slaught Memorial Papers No. 10 (February 1965), 21–28. Knuth gave a much more detailed exposition a decade later, in *Math. Comp.* **29** (1975), 121–136. Such methods can be considered as special cases of so-called "importance sampling"; see J. M. Hammersley and D. C. Handscomb, *Monte Carlo Methods*

Bernoulli
Tot tibi
depth-first search
Lucas
Trémaux
eight queens problem
Bezzel
Nauck
Gauss
Schumacher
lexicographically
Walker
Golomb
Baumert
commafree codes
optimization
lookahead
dynamic ordering
minimum remaining values
Wells
Bitner
Reingold
Gaschnig
backmarking
backjumping
Monte Carlo estimates
Hall
Knuth
importance sampling
Hammersley
Handscomb

(London: Methuen, 1964), 57–59. Studies of random self-avoiding walks such as the king paths discussed above were inaugurated by M. N. Rosenbluth and A. W. Rosenbluth, *J. Chemical Physics* **23** (1955), 356–359.

self-avoiding walks
king paths
Rosenbluth
Rosenbluth

EXERCISES

▶ **1.** [*22*] Explain how the tasks of generating (i) n-tuples, (ii) permutations, (iii) combinations, (iv) integer partitions, (v) set partitions, and (vi) nested parentheses can all be regarded as special cases of backtrack programming, by presenting suitable domains D_k and cutoff properties $P_l(x_1, \ldots, x_l)$ that satisfy (1) and (2).

2. [*10*] True or false: We can choose D_1 so that $P_1(x_1)$ is always true.

3. [*16*] Using a chessboard and eight coins to represent queens, one can follow the steps of Algorithm B and essentially traverse the tree of Fig. 68 by hand in about three hours. Invent a trick to save half of the work.

▶ **4.** [*20*] Reformulate Algorithm B as a *recursive* procedure called *try*(l), having global variables n and $x_1 \ldots x_n$, to be invoked by saying '*try*(1)'. Can you imagine why the author of this book decided *not* to present the algorithm in such a recursive form?

5. [*20*] Given r, with $1 \le r \le n$, in how many ways can 7 nonattacking queens be placed on an 8×8 chessboard, if no queen is placed in row r?

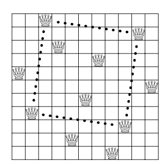
6. [*20*] (T. B. Sprague, 1890.) Are there any values $n > 5$ for which the n queens problem has a "framed" solution with $x_1 = 2$, $x_2 = n$, $x_{n-1} = 1$, and $x_n = n - 1$?

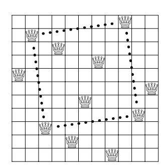
7. [*20*] Are there two 8-queen placements with the same $x_1x_2x_3x_4x_5x_6$?

8. [*21*] Can a $4m$-queen placement have $3m$ queens on "white" squares?

▶ **9.** [*22*] Adapt Algorithm W to the n queens problem, using bitwise operations on n-bit numbers as suggested in the text.

10. [*M25*] (W. Ahrens, 1910.) Both solutions of the n queens problem when $n = 4$ have *chiral symmetry*: Rotation by 90° leaves them unchanged, but reflection doesn't.





a) Can the n queens problem have a solution with reflection symmetry?
b) Show that chiral symmetry is impossible when $n \bmod 4 \in \{2, 3\}$.
c) Sometimes the solution to an n queens problem contains four queens that form the corners of a tilted square, as shown here. Prove that we can always get another solution by tilting the square the other way (but leaving the other $n - 4$ queens in place).
d) Let C_n be the number of chirally symmetric solutions, and suppose c_n of them have $x_k > k$ for $1 \le k \le n/2$. Prove that $C_n = 2^{\lfloor n/4 \rfloor} c_n$.

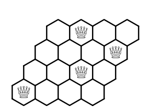
11. [*M28*] (*Wraparound queens.*) Replace (3) by the stronger conditions '$x_j \ne x_k$, $(x_k - x_j) \bmod n \ne k - j$, $(x_j - x_k) \bmod n \ne k - j$'. (The $n \times n$ grid becomes a torus.) Prove that the resulting problem is solvable if and only if n is not divisible by 2 or 3.

12. [*M30*] For which $n \ge 0$ does the n queens problem have at least one solution?

13. [*M25*] If exercise 11 has $T(n)$ toroidal solutions, show that $Q(mn) \ge Q(m)^n T(n)$.

14. [*HM47*] Does $(\ln Q(n))/(n \ln n)$ approach a positive constant as $n \to \infty$?

15. [*21*] Let $H(n)$ be the number of ways that n queen bees can occupy an $n \times n$ honeycomb so that no two are in the same line. (For example, one of the $H(4) = 7$ ways is shown here.) Compute $H(n)$ for small n.



16. [*15*] J. H. Quick (a student) noticed that the loop in step L2 of Algorithm L can be changed from 'while $x_l < 0$' to 'while $x_l \ne 0$', because x_l cannot be positive at that point of the algorithm. So he decided to eliminate the minus signs and just set $x_{l+k+1} \leftarrow k$ in step L3. Was it a good idea?

n-tuples
tuples
permutations
combinations
integer partitions
partitions
set partitions
nested parentheses
parentheses
domains
cutoff
properties
recursive
global variables
author
recursion versus iteration
Sprague
bitwise operations
Ahrens
chiral symmetry
Rotation by 90°
Wraparound queens
broken diagonal, see wraparound
torus
n queens problem
queen bees
bees
honeycomb
hexagons
Quick
Langford pairs+

17. [*17*] Suppose that $n = 4$ and Algorithm L has reached step L2 with $l = 4$ and $x_1x_2x_3 = 241$. What are the current values of $x_4x_5x_6x_7x_8$, $p_0p_1p_2p_3p_4$, and $y_1y_2y_3$?

19. [*M19*] What are the domains D_l in Langford's problem (7)?

▸ **20.** [*21*] Extend Algorithm L so that it forces $x_l \leftarrow k$ whenever $k \notin \{x_1, \ldots, x_{l-1}\}$.

▸ **21.** [*M25*] If $x = x_1x_2 \ldots x_{2n}$, let $x^D = (-x_{2n}) \ldots (-x_2)(-x_1) = -x^R$ be its dual.

a) Show that if n is odd and x solves Langford's problem (7), we have $x_k = n$ for some $k \le \lfloor n/2 \rfloor$ if and only if $x_k^D = n$ for some $k > \lfloor n/2 \rfloor$.

b) Find a similar rule that distinguishes x from x^D when n is even.

c) Consequently the algorithm of exercise 20 can be modified so that exactly one of each dual pair of solutions $\{x, x^D\}$ is visited.

22. [*M26*] Explore "loose Langford pairs": Replace '$j + k + 1$' in (7) by '$j + \lfloor 3k/2 \rfloor$'.

23. [*17*] We can often obtain one word rectangle from another by changing only a letter or two. Can you think of any 5×6 rectangles that almost match (10)?

24. [*20*] Customize Algorithm B so that it will find all 5×6 word rectangles.

▸ **25.** [*25*] Explain how to use *orthogonal lists*, as in Fig. 13 of Section 2.2.6, so that it's easy to visit all 5-letter words whose kth character is c, given $1 \le k \le 5$ and $\texttt{a} \le c \le \texttt{z}$. Use those sublists to speed up the algorithm of exercise 24.

26. [*21*] Can you find nice word rectangles of sizes 5×7, 5×8, 5×9, 5×10?

27. [*22*] What profile and average node costs replace (13) and (14) when we ask the algorithm of exercise 25 for 6×5 word rectangles instead of 5×6?

▸ **28.** [*23*] The method of exercises 24 and 25 does n levels of backtracking to fill the cells of an $m \times n$ rectangle one column at a time, using a trie to detect illegal prefixes in the rows. Devise a method that does mn levels of backtracking and fills just *one* cell per level, using tries for *both* rows and columns.

29. [*15*] What's the largest commafree subset of the following words?

`aced babe bade bead beef cafe cede dada dead deaf face fade feed`

▸ **30.** [*22*] Let w_1, w_2, …, w_m be four-letter words on an m-letter alphabet. Design an algorithm that accepts or rejects each w_j, according as w_j is commafree or not with respect to the accepted words of $\{w_1, \ldots, w_{j-1}\}$.

31. [*M22*] A two-letter block code on an m-letter alphabet can be represented as a digraph D on m vertices, with $a \to b$ if and only if ab is a codeword.

a) Prove that the code is commafree $\iff$ D has no oriented paths of length 3.

b) How many arcs can be in a digraph with no oriented paths of length r?

▸ **32.** [*M30*] (W. L. Eastman, 1965.) The following elegant construction yields a commafree code of maximum size for any *odd* block length n, over any alphabet. Given a sequence of $x = x_0x_1 \ldots x_{n-1}$ of nonnegative integers, where x differs from each of its other cyclic shifts $x_k \ldots x_{n-1}x_0 \ldots x_{k-1}$ for $0 < k < n$, the procedure outputs a cyclic shift σx with the property that the set of all such σx is commafree.

We regard x as an infinite periodic sequence $\langle x_n \rangle$ with $x_k = x_{k-n}$ for all $k \ge n$. Each cyclic shift then has the form $x_kx_{k+1} \ldots x_{k+n-1}$. The simplest nontrivial example occurs when $n = 3$, where $x = x_0x_1x_2x_0x_1x_2x_0 \ldots$ and we don't have $x_0 = x_1 = x_2$. In this case the algorithm outputs $x_kx_{k+1}x_{k+2}$ where $x_k > x_{k+1} \le x_{k+2}$; and the set of all such triples clearly satisfies the commafree condition.

domains
dual
loose Langford pairs
word rectangle
orthogonal lists
trie
commafree
two-letter block code
digraph
commafree
Eastman
aperiodic words
periodic sequence

One key idea is to think of x as partitioned into t substrings by boundary markers b_j, where $0 \le b_0 < b_1 < \cdots < b_{t-1} < n$ and $b_j = b_{j-t} + n$ for $j \ge t$. Then substring y_j is $x_{b_j} x_{b_j+1} \ldots x_{b_{j+1}-1}$. The number t of substrings is always odd. Initially $t = n$ and $b_j = j$ for all j; ultimately $t = 1$, and $\sigma x = y_0$ is the desired output.

Eastman's algorithm is based on comparison of adjacent substrings y_{j-1} and y_j. If those substrings have the same length, we use lexicographic comparison; otherwise we declare that the longer substring is bigger.

The second key idea is the notion of "dips," which are substrings of the form $z = z_1 \ldots z_k$ where $k \ge 2$ and $z_1 \ge \cdots \ge z_{k-1} < z_k$. It's easy to see that any string $y = y_0 y_1 \ldots$ in which we have $y_i < y_{i+1}$ for infinitely many i can be factored into a sequence of dips, $y = z^{(0)} z^{(1)} \ldots$, and this factorization is unique. For example,

$$3141592653589793238462643383\ldots = 314\ 15\ 926\ 535\ 89\ 79\ 323\ 846\ 26\ 4338\ 3\ldots.$$

Furthermore, if y is a periodic sequence, its factorization into dips is also ultimately periodic, although some of the initial factors may not occur in the period. For example,

$$123443550123443550123443550\ldots = 12\ 34\ 435\ 501\ 23\ 4435\ 501\ 23\ 4435\ \ldots.$$

Given a periodic, nonconstant sequence y described by boundary markers as above, where the period length t is odd, its periodic factorization will contain an odd number of odd-length dips. Each round of Eastman's algorithm simply retains the boundary points at the left of those odd-length dips. Then t is reset to the number of retained boundary points, and another round begins if $t > 1$.

a) Play through the algorithm by hand when $n = 19$ and $x = 3141592653589793238$.
b) Show that the number of rounds is at most $\lfloor \log_3 n \rfloor$.
c) Exhibit a binary x that achieves this worst-case bound when $n = 3^e$.
d) Implement the algorithm with full details. (It's surprisingly short!)
e) Explain why the algorithm yields a commafree code.

33. [*HM28*] What is the probability that Eastman's algorithm finishes in one round? (Assume that x is a random m-ary string of odd length $n > 1$, unequal to any of its other cyclic shifts. Use a generating function to express the answer.)

34. [*18*] Why can't a commafree code of length $(m^4 - m^2)/4$ contain 0001 and 2000?

▸ **35.** [*15*] Why do you think sequential data structures such as (16)–(23) weren't featured in Section 2.2.2 of this series of books (entitled "Sequential Allocation")?

36. [*17*] What's the significance of (a) `MEM[40d]`=`5e` and (b) `MEM[904]`=`84` in Table 1?

37. [*18*] Why is (a) `MEM[f8]` = `e7` and (b) `MEM[a0d]` = `ba` in Table 2?

38. [*20*] Suppose you're using the undoing scheme (26) and the operation $\sigma \leftarrow \sigma + 1$ has just bumped the current stamp σ to zero. What should you do?

▸ **39.** [*25*] Spell out the low-level implementation details of the candidate selection process in step C2 of Algorithm C. Use the routine store(a, v) of (26) whenever changing the contents of `MEM`, and use the following selection strategy:

a) Find a class c with the least number r of blue words.
b) If $r = 0$, set $x \leftarrow -1$; otherwise set x to a word in class c.
c) If $r > 1$, use the poison list to find an x that maximizes the number of blue words that could be killed on the other side of the prefix or suffix list that contains x.

▸ **40.** [*28*] Continuing exercise 39, spell out the details of step C3 when $x \ge 0$.

a) What updates should be done to `MEM` when a blue word x becomes red?
b) What updates should be done to `MEM` when a blue word x becomes green?

substrings
boundary markers
lexicographic comparison
dips
pi, as "random" data
worst-case bound
analysis of algs
Eastman
generating function
analysis of algs
data structures
Sequential Allocation
undoing scheme
bumped
stamp
poison list

c) Step C3 begins its job by making x green as in part (b). Explain how it should finish its job by updating the poison list.

42. [*M30*] Is there a *binary* ($m = 2$) commafree code with one codeword in each of the $(\sum_{d\backslash n}\mu(d)2^{n/d})/n$ cycle classes, for every word length n?

44. [*HM29*] A commafree code on m letters is equivalent to $2m!$ such codes if we permute the letters and/or replace each codeword by its left-right reflection.

Determine all of the nonisomorphic commafree codes of length 4 on m letters when m is (a) 2 (b) 3 (c) 4 and there are (a) 3 (b) 18 (c) 57 codewords.

45. [*M42*] Find a maximum-size commafree code of length 4 on $m = 5$ letters.

47. [*20*] Explain how the choices in Fig. 69 were determined from the "random" bits that are displayed. For instance, why was X_2 set to 1 in Fig. 69(b)?

48. [*M15*] Interpret the value $\mathrm{E}(D_1 \ldots D_l)$, in the text's Monte Carlo algorithm.

49. [*M22*] What's a simple martingale that corresponds to Theorem E?

▸ **50.** [*HM25*] Elmo uses Algorithm E with $D_k = \{1, \ldots, n\}$, $P_l = [x_1 > \cdots > x_l]$, $c = 1$.

a) Alice flips n coins independently, where coin k yields "heads" with probability $1/k$. True or false: She obtains exactly l heads with probability $\left[{n \atop l}\right]/n!$.

b) Let $Y_1, Y_2, \ldots, Y_l$ be the numbers on the coins that come up heads. (Thus $Y_1 = 1$, and $Y_2 = 2$ with probability 1/2.) Show that Pr(Alice obtains $Y_1, Y_2, \ldots, Y_l$) = Pr(Elmo obtains $X_1 = Y_l$, $X_2 = Y_{l-1}$, $\ldots$, $X_l = Y_1$).

c) Prove that Alice q.s. obtains at most $(\ln n)(\ln\ln n)$ heads.

d) Consequently Elmo's S is q.s. less than $\exp((\ln n)^2(\ln\ln n))$.

▸ **51.** [*M30*] Extend Algorithm B so that it also computes the minimum, maximum, mean, and variance of the Monte Carlo estimates S produced by Algorithm E.

52. [*M21*] Instead of choosing each y_i in step E5 with probability $1/d$, we could use a biased distribution where $\Pr(I = i \mid X_1, \ldots, X_{l-1}) = p_{X_1 \ldots X_{l-1}}(y_i) > 0$. How should the estimate S be modified so that its expected value in this general scheme is still $C()$?

53. [*M20*] If all costs $c(x_1, \ldots, x_l)$ are positive, show that the biased probabilities of exercise 52 can be chosen in such a way that the estimate S is always exact.

▸ **55.** [*M25*] The commafree code search procedure in Algorithm C doesn't actually fit the mold of Algorithm E, because it incorporates lookahead, dynamic ordering, reversible memory, and other enhancements to the basic backtrack paradigms. How could its running time be reliably estimated with Monte Carlo methods?

57. [*M20*] Algorithm E can potentially follow M different paths $X_1 \ldots X_{l-1}$ before it terminates, where M is the number of leaves of the backtrack tree. Suppose the final values of D at those leaves are $D^{(1)}, \ldots, D^{(M)}$. Prove that $(D^{(1)} \ldots D^{(M)})^{1/M} \ge M$.

58. [*27*] The text's king path problem is a special case of the general problem of counting simple paths from vertex s to vertex t in a given graph.

We can generate such paths by random walks from s that don't get stuck, if we maintain a table of values `DIST(`v`)` for all vertices v not yet in the path, representing the shortest distance from v to t through unused vertices. For with such a table we can simply move at each step to a vertex for which `DIST(`v`)` $< \infty$.

Devise a way to update the `DIST` table dynamically without unnecessary work.

59. [*26*] A ZDD with 3,174,197 nodes can be constructed for the family of all simple corner-to-corner king paths on a chessboard, using the method of exercise 7.1.4–225. Explain how to use this ZDD to compute (a) the total length of all paths; (b) the number of paths that touch any given subset of the center and/or corner points.

nonisomorphic
symmetries
"random" bits
Monte Carlo
martingale
coins
Stirling cycle numbers
q.s.
variance
biased distribution
commafree code
lookahead
dynamic ordering
reversible memory
simple paths
random walks
dynamic shortest distances
shortest distances, dynamic
ZDD
corner-to-corner
king paths
chessboard

▸ **60.** [*20*] Experiment with biased random walks (see exercise 52), weighting each non-dead-end king move to a new vertex v by $1 + \mathtt{DIST}(v)^2$ instead of choosing every such move with the same probability. Does this strategy improve on Fig. 70?

biased random walks
Woods
questionnaire

Table 666

TWENTY QUESTIONS (SEE EXERCISE 90)

1. The first question whose answer is A is:
(A) 1 (B) 2 (C) 3 (D) 4 (E) 5

2. The next question with the same answer as this one is:
(A) 4 (B) 6 (C) 8 (D) 10 (E) 12

3. The only two consecutive questions with identical answers are questions:
(A) 15 and 16 (B) 16 and 17 (C) 17 and 18 (D) 18 and 19 (E) 19 and 20

4. The answer to this question is the same as the answers to questions:
(A) 10 and 13 (B) 14 and 16 (C) 7 and 20 (D) 1 and 15 (E) 8 and 12

5. The answer to question 14 is:
(A) B (B) E (C) C (D) A (E) D

6. The answer to this question is:
(A) A (B) B (C) C (D) D (E) none of those

7. An answer that appears most often is:
(A) A (B) B (C) C (D) D (E) E

8. Ignoring answers that appear equally often, the least common answer is:
(A) A (B) B (C) C (D) D (E) E

9. The sum of all question numbers whose answers are correct and the same as this one is:
(A) $\in [59\,..\,62]$ (B) $\in [52\,..\,55]$ (C) $\in [44\,..\,49]$ (D) $\in [61\,..\,67]$ (E) $\in [44\,..\,53]$

10. The answer to question 17 is:
(A) D (B) B (C) A (D) E (E) wrong

11. The number of questions whose answer is D is:
(A) 2 (B) 3 (C) 4 (D) 5 (E) 6

12. The number of *other* questions with the same answer as this one is the same as the number of questions with answer:
(A) B (B) C (C) D (D) E (E) none of those

13. The number of questions whose answer is E is:
(A) 5 (B) 4 (C) 3 (D) 2 (E) 1

14. No answer appears exactly this many times:
(A) 2 (B) 3 (C) 4 (D) 5 (E) none of those

15. The set of odd-numbered questions with answer A is:
(A) {7} (B) {9} (C) not {11} (D) {13} (E) {15}

16. The answer to question 8 is the same as the answer to question:
(A) 3 (B) 2 (C) 13 (D) 18 (E) 20

17. The answer to question 10 is:
(A) C (B) D (C) B (D) A (E) correct

18. The number of prime-numbered questions whose answers are vowels is:
(A) prime (B) square (C) odd (D) even (E) zero

19. The last question whose answer is B is:
(A) 14 (B) 15 (C) 16 (D) 17 (E) 18

20. The maximum score that can be achieved on this test is:
(A) 18 (B) 19 (C) 20 (D) indeterminate
(E) achievable only by getting this question wrong

▸ **90.** [*M29*] (Donald R. Woods, 2000.) Find all ways to maximize the number of correct answers to the questionnaire in Table 666. Each question must be answered with a letter from A to E. *Hint:* Begin by clarifying the exact meaning of this exercise. What answers are best for the following two-question, two-letter "warmup problem"?

1. (A) Answer 2 is B. (B) Answer 1 is A.

2. (A) Answer 1 is correct. (B) Either answer 2 is wrong or answer 1 is A, but not both.

91. [*HM28*] Show that exercise 90 has a surprising, somewhat paradoxical answer if two changes are made to Table 666: 9(E) becomes '$\in [39\,..\,43]$'; 15(C) becomes '$\{11\}$'.

95. [*HM26*] Let P_n be the number of integer sequences $x_1 \ldots x_n$ such that $x_1 = 1$ and $1 \le x_{k+1} \le 2x_k$ for $1 \le k < n$. (The first few values are 1, 2, 6, 26, 166, 1626, ...; this sequence was introduced by A. Cayley in *Philosophical Magazine* (4) **13** (1857), 245–248, who showed that P_n enumerates the partitions of $2^n - 1$ into powers of 2.)

a) Show that P_n is the number of different profiles that are possible for a binary tree of height n.

b) Find an efficient way to compute P_n for large n. *Hint:* Consider the more general sequence $P_n^{(m)}$, defined similarly but with $x_1 = m$.

c) Use the estimation procedure of Theorem E to prove that $P_n \ge 2^{\binom{n}{2}}/(n-1)!$.

999. [*M00*] this is a temporary exercise (for dummies)

paradoxical
Cayley
binary partitions
partitions
profiles
height n

SECTION 7.2.2

1. Although many formulations are possible, the following may be the nicest: (i) D_k is arbitrary (but hopefully finite), and P_l is always true. (ii) $D_k = \{1, 2, \ldots, n\}$ and P_l = '$x_j \neq x_k$ for $1 \le j < k \le l$'. (iii) For combinations of n things from N, $D_k = \{1, \ldots, N+1-k\}$ and P_l = '$x_1 < \cdots < x_l$'. (iv) $D_k = \{0, 1, \ldots, \lfloor n/k \rfloor\}$; P_l = '$x_1 \ge \cdots \ge x_l$ and $n - (n-l)x_l \le x_1 + \cdots + x_l \le n$'. (v) For restricted growth strings, $D_k = \{0, \ldots, k-1\}$ and P_l = '$x_{j+1} \le 1 + \max(x_1, \ldots, x_j)$ for $1 \le j < l$'. (vi) For the indices of left parentheses (see 7.2.1.6–(8)), $D_k = \{1, \ldots, 2k-1\}$ and P_l = '$x_1 < \cdots < x_l$'.

2. True. (If not, set $D_1 \leftarrow D_1 \cap \{x \mid P_1(x)\}$.)

3. We can restrict D_1 to $\{1, 2, 3, 4\}$, because the reflection $(9-x_1)\ldots(9-x_8)$ of every solution $x_1 \ldots x_8$ is also a solution. (H. C. Schumacher made this observation in a letter to Gauss, 24 September 1850.) Notice that Fig. 68 is left-right symmetric.

4. $try(l)$ = "If $l > n$, visit $x_1 \ldots x_n$. Otherwise, for $x_l \leftarrow \min D_l$, $\min D_l + 1$, $\ldots$, $\max D_l$, if $P_l(x_1, \ldots, x_l)$ call $try(l+1)$."

This formulation is elegant, and fine for simple problems. But it doesn't give any clue about why the method is called "backtrack"! Nor does it yield efficient code for important problems whose inner loop is performed billions of times. We will see that the key to efficient backtracking is to provide good ways to update and downdate the data structures that speed up the testing of property P_l. The overhead of recursion can get in the way, and the actual iterative structure of Algorithm B isn't difficult to grasp.

5. Excluding cases with $j = r$ or $k = r$ from (3) yields respectively (312, 396, 430, 458, 458, 430, 396, 312) solutions. (With column r also omitted there are just (40, 46, 42, 80, 80, 42, 46, 40).)

6. Yes, probably for all $n > 16$. One such is $x_1 x_2 \ldots x_{17}$ = 2 17 12 10 7 14 3 5 9 13 15 4 11 8 6 1 16. [See *Proc. Edinburgh Math. Soc.* **8** (1890), 43 and Fig. 52.]

7. Yes: (42736815, 42736851); also therefore (57263148, 57263184).

8. Yes, at least when $m = 4$; e.g., $x_1 \ldots x_{16}$ = 5 8 13 16 3 7 15 11 6 2 10 14 1 4 9 12. There are no solutions when $m = 5$, but 7 10 13 20 17 24 3 6 23 11 16 21 4 9 14 2 19 22 1 8 5 12 15 18 works for $m = 6$. (Are there solutions for all even $m \ge 4$? C. F. de Jaenisch, *Traité des applications de l'analyse mathématique au jeu des échecs* **2** (1862), 132–133, noted that all 8-queen solutions have four of each color. He proved that the number of white queens must be even, because $\sum_{k=1}^{4m}(x_k + k)$ is even.)

9. Let bit vectors a_l, b_l, c_l represent the "useful" elements of the sets in (6), with $a_l = \sum\{2^{x-1} \mid x \in A_l\}$, $b_l = \sum\{2^{x-1} \mid x \in B_l \cap [1\,..\,n]\}$, $c_l = \sum\{2^{x-1} \mid x \in C_l \cap [1\,..\,n]\}$. Then step W2 sets $s_l \leftarrow \mu \mathbin{\&} \bar{a}_l \mathbin{\&} \bar{b}_l \mathbin{\&} \bar{c}_l$, where μ is the mask $2^n - 1$.

In step W3 we can set $t \leftarrow s_l \mathbin{\&} (-s_l)$, $a_l \leftarrow a_{l-1} + t$, $b_l \leftarrow (b_{l-1} + t) \gg 1$, $c_l \leftarrow ((c_{l-1} + t) \ll 1) \mathbin{\&} \mu$; and it's also convenient to set $s_l \leftarrow s_l - t$ at this time, instead of deferring this change to step W4.

(There's no need to store x_l in memory, or even to compute x_l in step W3 as an integer in $[1\,..\,n]$, because x_l can be deduced from $a_l - a_{l-1}$ when a solution is found.)

10. (a) Only when $n = 1$, because reflected queens can capture each other.

(b) Queens not in the center must appear in groups of four.

(c) The four queens occupy the same rows, columns, and diagonals in both cases.

(d) In each solution counted by c_n we can independently tilt (or not) each of the $\lfloor n/4 \rfloor$ groups of four. [*Mathematische Unterhaltungen und Spiele* **1**, second edition (Leipzig: Teubner, 1910), 249–258.]

restricted growth strings
reflection
Schumacher
Gauss
symmetric
inner loop
backtracking, efficient
iteration versus recursion
de Jaenisch
mask

11. Suppose the x_k are distinct. Then $\sum_{k=1}^{n}(x_k+k) = 2\binom{n+1}{2} \equiv 0$ (modulo n). If the numbers $(x_k+k) \bmod n$ are also distinct, we have also $\sum_{k=1}^{n} k \equiv \binom{n+1}{2}$. But that is impossible when n is even.

Now suppose further that the numbers $(x_k-k) \bmod n$ are distinct. Then we have $\sum_{k=1}^{n}(x_k+k)^2 \equiv \sum_{k=1}^{n}(x_k-k)^2 \equiv \sum_{k=1}^{n} k^2 = n(n+1)(2n+1)/6$. And we also have $\sum_{k=1}^{n}(x_k+k)^2 + \sum_{k=1}^{n}(x_k-k)^2 = 4n(n+1)(2n+1)/6 \equiv 2n/3$, which is impossible when n is a multiple of 3. [See W. Ahrens, *Mathematische Unterhaltungen und Spiele* **2**, second edition (1918), 364–366, where G. Pólya cites a more general result of A. Hurwitz that applies to wraparound diagonals of other slopes.]

Conversely, if n isn't divisible by 2 or 3, we can let $x_n = n$ and $x_k = (2k) \bmod n$ for $1 \le k < n$. (The rule $x_k = (3k) \bmod n$ also works. See Édouard Lucas, *Récréations Mathématiques* **1** (1882), 84–86.)

12. The $(n+1)$ queens problem clearly has a solution with a queen in a corner if and only if the n queens problem has a solution with a queen-free main diagonal. Hence by the previous answer there's always a solution when $n \bmod 6 \in \{0,1,4,5\}$.

Another nice solution was found by J. Franel [*L'Intermédiaire des Mathématiciens* **1** (1894), 140–141] when $n \bmod 6 \in \{2,4\}$: Let $x_k = \bigl(n/2 + 2k - 3[2k \le n]\bigr) \bmod n + 1$, for $1 \le k \le n$. With this setup we find that $x_k - x_j = \pm(k-j)$ and $1 \le j < k \le n$ implies $(1 \text{ or } 3)(k-j) + (0 \text{ or } 3) \equiv 0$ (modulo n); hence $k - j = n - (1 \text{ or } 3)$. But the values of x_1, x_2, x_3, x_{n-2}, x_{n-1}, x_n give no attacking queens except when $n=2$.

Franel's solution has empty diagonals, so it provides solutions also for $n \bmod 6 \in \{3,5\}$. We conclude that only $n=2$ and $n=3$ are impossible.

[A more complicated construction for all $n > 3$ had been given earlier by E. Pauls, in *Deutsche Schachzeitung* **29** (1874), 129–134, 257–267. Pauls also explained how to find all solutions, in principle, by building the tree level by level (*not* backtracking).]

13. For $1 \le j \le n$, let $x_1^{(j)} \ldots x_m^{(j)}$ be a solution for m queens, and let $y_1 \ldots y_n$ be a solution for n toroidal queens. Then $X_{(i-1)n+j} = (x_i^{(j)} - 1)n + y_j$ (for $1 \le i \le m$ and $1 \le j \le n$) is a solution for mn queens. [I. Rivin, I. Vardi, and P. Zimmermann, *AMM* **101** (1994), 629–639, Theorem 2.]

14. [Rivin, Vardi, and Zimmermann, in the paper just cited, observe that in fact the sequence $(\ln Q(n))/(n \ln n)$ appears to be *increasing*.]

15. Let the queen in row k be in cell k. Then we have a "relaxation" of the n queens problem, with $|x_k - x_j|$ becoming just $x_k - x_j$ in (3); so we can ignore the b vector in Algorithm B* or in exercise 9. We get

$n =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$H(n) =$	1	1	1	3	7	23	83	405	2113	12657	82297	596483	4698655	40071743	367854835

[N. J. Cavenagh and I. M. Wanless, *Discr. Appl. Math.* **158** (2010), 136–146, Table 2.]

16. It fails spectacularly in step L5. The minus signs, which mark decisions that were previously forced, are crucial tags for backtracking.

17. $x_4 \ldots x_8 = \bar{2}\bar{1}0\bar{4}0$, $p_0 \ldots p_4 = 33300$, and $y_1y_2y_3 = 130$. (If $x_i \le 0$ the algorithm will never look at y_i; hence the current state of $y_4 \ldots y_8$ is irrelevant. But y_4y_5 happens to be 20, because of past history; y_6, y_7, and y_8 haven't yet been touched.)

19. We could say D_l is $\{-n, \ldots, -2, -1, 1, 2, \ldots, n\}$, or $\{k \mid k \ne 0 \text{ and } 2 - l \le k \le 2n - l - 1\}$, or anything in between. (But this observation isn't very useful.)

Ahrens
Pólya
Hurwitz
Lucas
Franel
Pauls
tree
breadth first search
Rivin
Vardi
Zimmermann
semi-queens
Cavenagh
Wanless

parity argument
trie
signature
trie
bitwise AND

20. First we add a Boolean array $a_1 \ldots a_n$, where a_k means "k has appeared," as in Algorithm B*. It's $0\ldots0$ in step L1; we set $a_k \leftarrow 1$ in step L3, $a_k \leftarrow 0$ in step L5.

The loop in step L2 becomes "while $x_l < 0$, go to L5 if $l \geq n-1$ and $a_{2n-l-1} = 0$, otherwise set $l \leftarrow l+1$." After finding $l+k+1 \leq 2n$ in L3, and before testing x_{l+k+1} for 0, insert this: "If $l \geq n-1$ and $a_{2n-l-1} = 0$, while $l+k+1 \neq 2n$ set $j \leftarrow k$, $k \leftarrow p_k$."

21. (a) In any solution $x_k = n \iff x_{k+n+1} = -n \iff x^D_{n-k} = n$.

(b) $x_k = n-1$ for some $k \leq n/2$ if and only if $x^D_k = n-1$ for some $k > n/2$.

(c) Let $n' = n - [n \text{ is even}]$. Change '$l \geq n-1$ and $a_{2n-l-1} = 0$' in the modified step L2 to '$(l = \lfloor n/2 \rfloor$ and $a_{n'} = 0)$ or $(l \geq n-1$ and $a_{2n-l-1} = 0)$'. Insert the following before the other insertion into step L3: "If $l = \lfloor n/2 \rfloor$ and $a_{n'} = 0$, while $k \neq n'$ set $j \leftarrow k$, $k \leftarrow p_k$." And in step L5 — this subtle detail is needed when n is even — go to L5 instead of L4 if $l = \lfloor n/2 \rfloor$ and $k = n'$.

22. The solutions $1\bar{1}$ and $21\bar{1}\bar{2}$ for $n = 1$ and $n = 2$ are self-dual; the solutions for $n = 4$ and $n = 5$ are $431\bar{1}2\bar{3}\bar{4}\bar{2}$, $245\bar{2}31\bar{1}\bar{4}\bar{3}\bar{5}$, $451\bar{1}23\bar{4}\bar{2}\bar{5}\bar{3}$, and their duals. The total number of solutions for $n = 1, 2, \ldots$ is 1, 1, 0, 2, 4, 20, 0, 156, 516, 2008, 0, 52536, 297800, 1767792, 0, 75678864, ...; there are none when $n \bmod 4 = 3$, by a parity argument.

Algorithm L needs only obvious changes. To compute solutions by a streamlined method like exercise 21, use $n' = n - (0, 1, 2, 0)$ and substitute '$l = \lfloor n/4 \rfloor + (0, 1, 2, 1)$' for '$l = \lfloor n/2 \rfloor$', when $n \bmod 4 = (0, 1, 2, 3)$; also replace '$l \geq n-1$ and $a_{2n-l-1} = 0$' by '$l \geq \lceil n/2 \rceil$ and $a_{\lfloor (4n+2-2l)/3 \rfloor} = 0$'. The case $n = 15$ is proved impossible with 397 million nodes and 9.93 gigamems.

23. `slums` $\to$ `sluff`, `slump`, `slurs`, `slurp`, or `sluts`; (`slums`, `total`) $\to$ (`slams`, `tonal`).

24. Build the list of 5-letter words and the trie of 6-letter words in step B1; also set $a_{01}a_{02}a_{03}a_{04}a_{05} \leftarrow 00000$. Use $\min D_l = 1$ in step B2 and $\max D_l = 5757$ in step B4. Testing P_l in step B3, if word x_3 is $c_1c_2c_3c_4c_5$, consists of forming $a_{l1} \ldots a_{l5}$, where $a_{lk} = \mathit{trie}[a_{(l-1)k}, c_k]$ for $1 \leq k \leq 5$; but jump to B4 if any a_{lk} is zero.

25. There are 5×26 singly linked lists, accessed from pointers h_{kc}, all initially zero. The xth word $c_{x1}c_{x2}c_{x3}c_{x4}c_{x5}$, for $1 \leq x \leq 5757$, belongs to 5 lists and has five pointers $l_{x1}l_{x2}l_{x3}l_{x4}l_{x5}$. To insert it, set $l_{xk} \leftarrow h_{kc_{xk}}$, $h_{kc_{xk}} \leftarrow x$, and $s_{kc_{xk}} \leftarrow s_{kc_{xk}} + 1$, for $1 \leq k \leq 5$. (Thus s_{kc} will be the length of the list accessed from h_{kc}.)

We can store a "signature" $\sum_{c=1}^{26} 2^{c-1}[\mathit{trie}[a, c] \neq 0]$ with each node a of the trie. For example, the signature for node 260 is $2^0+2^4+2^8+2^{14}+2^{17}+2^{20}+2^{24} = {}^\#\mathtt{1124111}$, according to (11); here $\mathtt{a} \leftrightarrow 1, \ldots, \mathtt{z} \leftrightarrow 26$.

The process of running through all x that match a given signature y with respect to position z, as needed in steps B2 and B4, now takes the following form: (i) Set $i \leftarrow 0$. (ii) While $2^i \mathbin{\&} y = 0$, set $i \leftarrow i+1$. (iii) Set $x \leftarrow h_{z(i+1)}$; go to (vi) if $x = 0$. (iv) Visit x. (v) Set $x \leftarrow l_{xz}$; go to (iv) if $x \neq 0$. (vi) Set $i \leftarrow i+1$; go to (ii) if $2^i \leq y$.

Let $\mathit{trie}[a, 0]$ be the signature of node a. We choose z and $y = \mathit{trie}[a_{(l-1)z}, 0]$ in step B2 so that the number of nodes to visit, $\sum_{c=1}^{26} s_{zc}[2^{c-1} \mathbin{\&} y \neq 0]$, is minimum for $1 \leq z \leq 5$. For example, when $l = 3$, $x_1 = 1446$, and $x_2 = 185$ as in (10), that sum for $z = 1$ is $s_{11}+s_{15}+s_{19}+s_{1(15)}+s_{1(18)}+s_{1(21)}+s_{1(25)} = 296+129+74+108+268+75+47 = 997$; and the sums for $z = 2, 3, 4, 5$ are 4722, 1370, 5057, and 1646. Hence we choose $z = 1$ and $y = {}^\#\mathtt{1124111}$; only 997 words, not 5757, need be tested for x_3.

The values y_l and z_l are maintained for use in backtracking. (In practice we keep x, y, and z in registers during most of the computation. Then we set $x_l \leftarrow x$, $y_l \leftarrow y$, $z_l \leftarrow z$ before increasing $l \leftarrow l+1$ in step B3; and we set $x \leftarrow x_l$, $y \leftarrow y_l$, $z \leftarrow z_l$ in

author
Gattegno
compressed trie
trie, compressed
saturating ternary addition

step B5. We also keep i in a register, while traversing the sublists as above; this value is restored in step B5 by setting it to the zth letter of word x, decreased by `'a'`.)

26. Here are the author's favorite 5×7 and 5×8, and the *only* 5×9's:

`smashes`	`grandest`	`pastelist`	`varistors`
`partial`	`renounce`	`accidence`	`agentival`
`immense`	`episodes`	`mortgagor`	`coelomate`
`emerged`	`basement`	`proreform`	`undeleted`
`sadness`	`eyesores`	`andesytes`	`oysterers`

No 5×10 word rectangles exist, according to our ground rules.

27. (1, 15727, 8072679, 630967290 90962081, 625415) and (15727.0, 4321.6, 1749.7, 450.4, 286.0). Total time $\approx$ 18.3 teramems. (In Section 7.2.2.1 we'll study a method that is symmetrical between rows and columns.)

28. Build a separate trie for the m-letter words; but instead of having trie nodes of size 26 as in (11), it's better to convert this trie into a *compressed* representation that omits the zeros. For example, the compressed representation of the node for prefix '`corne`' in (12) consists of five consecutively stored pairs of entries ('`a`', 3879), ('`d`', 3878), ('`l`', 9602), ('`r`', 171), ('`t`', 5013), followed by $(0, 0)$. Similarly, each shorter prefix with c descendants is represented by c consecutive pairs (character, link), followed by $(0, 0)$ to mark the end of the node. Steps B3 and B4 are now very convenient.

Level l corresponds to row $i_l = 1 + (l-1) \bmod m$ and column $j_l = 1 + \lfloor (l-1)/m \rfloor$. For backtracking we store the n-trie pointer a_{i_l, j_l} as before, together with an index x_l into the compressed m-trie.

This method was suggested by Bernard Gattegno in 1996 (unpublished). It finds all 5×6 word rectangles in just 400 gigamems; and its running time for "transposed" 6×5 rectangles turns out to be slightly less (380 gigamems). Notice that only one mem is needed to access each (character, link) pair in the compressed trie.

29. Leave out `face` and (of course) `dada`; the remaining eleven are fine.

30. Keep tables p_i, p'_{ij}, p''_{ijk}, s_i, s'_{ij}, s''_{ijk}, for $0 \le i, j, k < m$, each capable of storing a ternary digit. Also keep a table x_0, x_1, ... of tentatively accepted words. Begin with $g \leftarrow 0$. Then for each input $w_j = abcd$, where $0 \le a, b, c, d < m$, set $x_g \leftarrow abcd$ and also do the following: Set $p_a \leftarrow p_a \dot{+} 1$, $p'_{ab} \leftarrow p'_{ab} \dot{+} 1$, $p''_{abc} \leftarrow p''_{abc} \dot{+} 1$, $s_d \leftarrow s_d \dot{+} 1$, $s'_{cd} \leftarrow s'_{cd} \dot{+} 1$, $s''_{bcd} \leftarrow s''_{bcd} \dot{+} 1$, where $x \dot{+} y = \min(2, x + y)$ denotes saturating ternary addition. Then if $s_{a'}p''_{b'c'd'} + s'_{a'b'}p'_{c'd'} + s''_{a'b'c'}p_{d'} = 0$ for all $x_k = a'b'c'd'$, where $0 \le k \le g$, set $g \leftarrow g + 1$. Otherwise reject w_j and set $p_a \leftarrow p_a - 1$, $p'_{ab} \leftarrow p'_{ab} - 1$, $p''_{abc} \leftarrow p''_{abc} - 1$, $s_d \leftarrow s_d - 1$, $s'_{cd} \leftarrow s'_{cd} - 1$, $s''_{bcd} \leftarrow s''_{bcd} - 1$.

31. (a) The word bc appears in message $abcd$ if and only if $a \to b$, $b \to c$, and $c \to d$.

(b) For $0 \le k < r$, put vertex v into class k if the longest path from v has length k. Given any such partition, we can include all arcs from class k to class $j < k$ without increasing the path lengths. So it's a question of finding the maximum of $\sum_{0 \le j < k < r} p_j p_k$ subject to $p_0 + p_1 + \cdots + p_{r-1} = m$. The values $p_j = \lfloor (m+j)/r \rfloor$ achieve this (see exercise 7.2.1.4–68(a)). When $r = 3$ the maximum simplifies to $\lfloor m^2/3 \rfloor$.

32. (a) The factors of the period, 15 926 535 89 79 323 8314, begin at the respective boundary points 3, 5, 8, 11, 13, 15, 18 (and then 3 + 19 = 22, etc.). Thus round 1 retains boundaries 5, 8, and 15. The second-round substrings $y_0 = 926$, $y_1 = 5358979$, $y_2 = 323831415$ have different lengths, so lexicographic comparison is unnecessary; the answer is $y_2 y_0 y_1 = x_{15} \ldots x_{33}$.

(b) Each substring consists of at least three substrings of the previous round.

superdip
Eastman
Scholtz
aperiodic

(c) Let $a_0 = 0$, $b_0 = 1$, $a_{e+1} = a_e a_e b_e$, $b_{e+1} = a_e b_e b_e$; use a_e or b_e when $n = 3^e$.

(d) We use an auxiliary subroutine 'less(i)', which returns $[y_{i-1} < y_i]$, given $i > 0$: If $b_i - b_{i-1} \neq b_{i+1} - b_i$, return $[b_i - b_{i-1} < b_{i+1} - b_i]$. Otherwise, for $j = 0, 1, \ldots,$ while $b_i + j < b_{i+1}$, if $x_{b_{i-1}+j} \neq x_{b_i+j}$ return $[x_{b_{i-1}+j} < x_{b_i+j}]$. Otherwise return 0.

The tricky part of the algorithm is to discard initial factors that aren't periodic. The secret is to let i_0 be the smallest index such that $y_{i-3} \geq y_{i-2} < y_{i-1}$; then we can be sure that a factor begins with y_i.

O1. [Initialize.] Set $x_j \leftarrow x_{j-n}$ for $n \leq j < 2n$, $b_j \leftarrow j$ for $0 \leq j < 2n$, and $t \leftarrow n$.

O2. [Begin a round.] Set $t' \leftarrow 0$. Find the smallest $i > 0$ such that less$(i) = 0$. Then find the smallest $j \geq i + 2$ such that less$(j - 1) = 1$ and $j \leq t + 2$. (If no such j exists, report an error: The input x was equal to one of its cyclic shifts.) Set $i \leftarrow i_0 \leftarrow j \bmod t$. (Now a dip of the period begins at i_0.)

O3. [Find the next factor.] Find the smallest $j \geq i + 2$ such that less $(j - 1) = 1$. If $j - i$ is even, go to O5.

O4. [Retain a boundary.] If $j < t$, set $b'_{t'} \leftarrow b_j$; otherwise set $b'_k \leftarrow b'_{k-1}$ for $t' \geq k > 0$ and $b'_0 \leftarrow b_{j-t}$. Finally set $t' \leftarrow t' + 1$.

O5. [Done with round?] If $j < i_0 + t$, set $i \leftarrow j$ and return to O3. Otherwise, if $t' = 1$, terminate; σx begins at item $x_{b'_0}$. Otherwise set $t \leftarrow t'$, $b_k \leftarrow b'_k$ for $0 \leq k < t$, and $b_k \leftarrow b_{k-t} + n$ for $k \geq t$ while $b_{k-t} < n$. Return to O2. ▮

(e) Say that a "superdip" is a dip of odd length followed by zero or more dips of even length. Any infinite sequence y that begins with an odd-length dip has a unique factorization into superdips. Those superdips can, in turn, be regarded as atomic elements of a higher-level string that can be factored into dips. The result σx of Algorithm O is an infinite periodic sequence that allows repeated factorization into infinite periodic sequences of superdips at higher and higher levels, until becoming constant.

Notice that the first dip of σx ends at position i_0 in the algorithm, because its length isn't 2. Therefore we can prove the commafree property by observing that, if codeword $\sigma x''$ appears within the concatenation $\sigma x\, \sigma x'$ of two codewords, its superdip factors are also superdip factors of those codewords. This yields a contradiction if any of σx, $\sigma x'$, or $\sigma x''$ is a superdip. Otherwise the same observation applies to the superdip factors at the next level. [Eastman's original algorithm was essentially the same, but presented in a more complicated way; see *IEEE Transactions* **IT-11** (1965), 263–267. R. A. Scholtz subsequently discovered an interesting and totally different way to define exactly the same set of codewords, in *IEEE Transactions* **IT-15** (1965), 300–306.]

33. Let $f_k(m)$ be the number of dips of length k for which $m > z_1$ and $z_k < m$. The number of such sequences with $z_2 = j$ is $(m - j - 1)\binom{m-j+k-3}{k-2} = (k-1)\binom{m-j+k-3}{k-1}$; summing for $0 \leq j < m$ gives $f_k(m) = (k-1)\binom{m+k-2}{k}$. Thus $F_m(z) = \sum_{k=0}^{\infty} f_k(m) z^k = (mz-1)/(1-z)^m$. (The fact that $f_0(m) = -1$ in these formulas turns out to be useful!)

Algorithm O finishes in one round if and only if some cyclic shift of x is a superdip. The number of aperiodic x that finish in one round is therefore $n[z^n]\,G_m(z)$, where

$$G_m(z) = \frac{F_m(-z) - F_m(z)}{F_m(-z) + F_m(z)} = \frac{(1+mz)(1-z)^m - (1-mz)(1+z)^m}{(1+mz)(1-z)^m + (1-mz)(1+z)^m}.$$

To get the stated probability, divide by $\sum_{d\backslash n} \mu(d) m^{n/d}$, the number of aperiodic x. (See Eq. 7.2.1.1–(60). For $n = 3, 5, 7, 9$ these probabilities are 1, 1, 1, and $1 - 3/\binom{m^3-1}{3}$.)

34. If so, it couldn't have 0011, 0110, 1100, or 1001.

35. That section considered such representations of stacks and queues, but not of unordered sets, because large blocks of sequential memory were either nonexistent or ultra-expensive in olden days. Linked lists were the only decent option for families of variable-size sets, because they could more readily fit in a limited high-speed memory.

stacks
queues
memory constraints, historic
unordered sets
Linked lists
deletion

36. (a) The blue word x with $\alpha = \mathtt{d}$ (namely 1101) appears in its `P2` list at location `5e`.

(b) The `P3` list for words of the form 010∗ is empty. (Both 0100 and 0101 are red.)

37. (a) The `S2` list of 0010 has become closed (hence 0110 and 1110 are hidden).

(b) Word 1101 moved to the former position of 1001 in its `S1` list, when 1001 became red. (Previously 1011 had moved to the former position of 0001.)

38. In this case, which of course happens rarely, it's safe to set all elements of `STAMP` to zero and set $\sigma \leftarrow 1$. (Do *not* be tempted to save one line of code by setting all `STAMP` elements to -1 and leaving $\sigma = 0$. That might fail when σ reaches the value -1!)

39. (a) Set $r \leftarrow m + 1$. Then for $k \leftarrow 0, 1, \ldots, f-1$, set $t \leftarrow \mathtt{FREE[}k\mathtt{]}$, $j \leftarrow \mathtt{MEM[CLOFF} + 4t + m^4\mathtt{]} - (\mathtt{CLOFF} + 4t)$, and if $j < r$ set $r \leftarrow j$, $c \leftarrow t$; break out of the loop if $r = 0$.

(b) If $r > 0$ set $x \leftarrow \mathtt{MEM[CLOFF} + 4cl(\mathtt{ALF[}x\mathtt{]})\mathtt{]}$.

(c) If $r > 1$ set $q \leftarrow 0$, $p' \leftarrow \mathtt{MEM[PP]}$, and $p \leftarrow \mathtt{POISON}$. While $p < p'$ do the following steps: Set $y \leftarrow \mathtt{MEM[}p\mathtt{]}$, $z \leftarrow \mathtt{MEM[}p+1\mathtt{]}$, $y' \leftarrow \mathtt{MEM[}y + m^4\mathtt{]}$, and $z' \leftarrow \mathtt{MEM[}z + m^4\mathtt{]}$. (Here y and z point to the heads of prefix or suffix lists; y' and z' point to the tails.) If $y = y'$ or $z = z'$, delete entry p from the poison list; this means, as in (18), to set $p' \leftarrow p'-2$, and if $p \neq p'$ to store$(p, \mathtt{MEM[}p'\mathtt{]})$ and store$(p+1, \mathtt{MEM[}p'+1\mathtt{]})$. Otherwise set $p \leftarrow p+2$; if $y'-y \geq z'-z$ and $y'-y > q$, set $q \leftarrow y'-y$ and $x \leftarrow \mathtt{MEM[}z\mathtt{]}$; if $y'-y < z'-z$ and $z'-z > q$, set $q \leftarrow z'-z$ and $x \leftarrow \mathtt{MEM[}y\mathtt{]}$. Finally, after p has become equal to p', store$(\mathtt{PP}, p')$ and set $c \leftarrow cl(\mathtt{ALF[}x\mathtt{]})$. (Experiments show that this "max kill" strategy for $r > 1$ slightly outperforms a selection strategy based on r alone.)

40. (a) First there's a routine rem(α, δ, o) that removes an item from a list, following the protocol (21): Set $p \leftarrow \delta + o$ and $q \leftarrow \mathtt{MEM[}p + m^4\mathtt{]} - 1$. If $q \geq p$ (meaning that list p isn't closed or being killed), store$(p + m^4, q)$, set $t \leftarrow \mathtt{MEM[}\alpha + o - m^4\mathtt{]}$; and if $t \neq q$ also set $y \leftarrow \mathtt{MEM[}q\mathtt{]}$, store$(t, y)$, and store$(\mathtt{ALF[}y\mathtt{]} + o - m^4, t)$.

Now, to redden x we set $\alpha \leftarrow \mathtt{ALF[}x\mathtt{]}$, store$(\alpha, \mathtt{RED})$; then rem$(\alpha, p_1(\alpha), \mathtt{P1OFF})$, rem$(\alpha, p_2(\alpha), \mathtt{P2OFF})$, ..., rem$(\alpha, s_3(\alpha), \mathtt{S3OFF})$, and rem$(\alpha, 4cl(\alpha), \mathtt{CLOFF})$.

(b) A simple routine close(δ, o) closes list $\delta+o$: Set $p \leftarrow \delta+o$ and $q \leftarrow \mathtt{MEM[}p + m^4\mathtt{]}$; if $q \neq p-1$, store$(p + m^4, p-1)$.

Now, to green x we set $\alpha \leftarrow \mathtt{ALF[}x\mathtt{]}$, store$(\alpha, \mathtt{GREEN})$; then close$(p_1(\alpha), \mathtt{P1OFF})$, close$(p_2(\alpha), \mathtt{P2OFF})$, ..., close$(s_3(\alpha), \mathtt{S3OFF})$, and close$(4cl(\alpha), \mathtt{CLOFF})$. Finally, for $p \leq r < q$ (using the p and q that were just set within 'close'), if $\mathtt{MEM[}r\mathtt{]} \neq x$ redden $\mathtt{MEM[}r\mathtt{]}$.

(c) First set $p' \leftarrow \mathtt{MEM[PP]} + 6$, and store$(p'-6, p_1(\alpha) + \mathtt{S1OFF})$, store$(p'-5, s_3(\alpha) + \mathtt{P3OFF})$, store$(p'-4, p_2(\alpha) + \mathtt{S2OFF})$, store$(p'-3, s_2(\alpha) + \mathtt{P2OFF})$, store$(p'-2, p_3(\alpha) + \mathtt{S3OFF})$, store$(p'-1, s_1(\alpha) + \mathtt{P1OFF})$; this adds the three poison items (27).

Then set $p \leftarrow \mathtt{POISON}$ and do the following while $p < p'$: Set y, z, y', z' as in answer 39(c), and delete poison entry p if $y = y'$ or $z = z'$. Otherwise if $y' < y$ and $z' < z$, go to C6 (a poisoned suffix-prefix pair is present). Otherwise if $y' > y$ and $z' > z$, set $p \leftarrow p + 2$. Otherwise if $y' < y$ and $z' > z$, store$(z + m^4, z)$, redden $\mathtt{MEM[}r\mathtt{]}$ for $z \leq r < z'$, and delete poison entry p. Otherwise (namely if $y' > y$ and $z' < z$), store$(y + m^4, y)$, redden $\mathtt{MEM[}r\mathtt{]}$ for $y \leq r < y'$, and delete poison entry p.

Finally, after p has become equal to p', store$(\mathtt{PP}, p')$.

42. Exercise 32 exhibits such codes explicitly for all odd n. The earliest papers on the subject gave solutions for $n = 2, 4, 6, 8$. Yoji Niho subsequently found a code for $n = 10$ but was unable to resolve the case $n = 12$ [*IEEE Trans.* **IT-19** (1973), 580–581].

This problem can readily be encoded in CNF and given to a SAT solver. The case $n = 10$ involves 990 variables and 8.6 million clauses, and is solved by Algorithm 7.2.2.2C in 10.5 gigamems. The case $n = 12$ involves 4020 variables and 175 million clauses. After being split into seven independent subproblems (by appending mutually exclusive unit clauses), it was proved *unsatisfiable* by that algorithm after about 86 teramems of computation.

So the answer is "No." The maximum-size code for $n = 12$ remains unknown.

44. (a) There are 28 commafree binary codes of size 3 and length 4; Algorithm C produces half of them, because it assumes that cycle class [0001] is represented by 0001 or 0010. They form eight equivalence classes, two of which are symmetric under the operation of complementation-and-reflection; representatives are $\{0001, 0011, 0111\}$ and $\{0010, 0011, 1011\}$. The other six are represented by $\{0001, 0110, 0111 \text{ or } 1110\}$, $\{0001, 1001, 1011 \text{ or } 1101\}$, $\{0001, 1100, 1101\}$, $\{0010, 0011, 1101\}$.

(b) Algorithm C produces half of the 144 solutions, which form twelve equivalence classes. Eight are represented by {0001, 0002, 1001, 1002, 2201, 2001, 2002, 2011, 2012, 2102, 2112, 2122 or 2212} and ($\{0102, 1011, 1012\}$ or $\{1020, 1101, 2101\}$) and ($\{1202, 2202, 2111\}$ or $\{2021, 2022, 1112\}$); four are represented by {0001, 0020, 0021, 0022, 1001, 1020, 1021, 1022, 1121 or 1211, 1201, 1202, 1221, 2001, 2201, 2202} and ($\{1011, 1012, 2221\}$ or $\{1101, 2101, 1222\}$).

(c) Algorithm C yields half of the 2304 solutions, which form 48 equivalence classes. Twelve classes have unique representatives that omit cycle classes [0123], [0103], [1213], one such being the code {0010, 0020, 0030, 0110, 0112, 0113, 0120, 0121, 0122, 0130, 0131, 0132, 0133, 0210, 0212, 0213, 0220, 0222, 0230, 0310, 0312, 0313, 0320, 0322, 0330, 0332, 0333, 1110, 1112, 1113, 2010, 2030, 2110, 2112, 2113, 2210, 2212, 2213, 2230, 2310, 2312, 2313, 2320, 2322, 2330, 2332, 2333, 3110, 3112, 3113, 3210, 3212, 3213, 3230, 3310, 3312, 3313}. The others each have two representatives that omit classes [0123], [0103], [0121], one such being the code {0001, 0002, 0003, 0201, 0203, 1001, 1002, 1003, 1011, 1013, 1021, 1022, 1023, 1031, 1032, 1033, 1201, 1203, 1211, 1213, 1221, 1223, 1231, 1232, 1233, 1311, 1321, 1323, 1331, 2001, 2002, 2003, 2021, 2022, 2023, 2201, 2203, 2221, 2223, 3001, 3002, 3003, 3011, 3013, 3021, 3022, 3023, 3031, 3032, 3033, 3201, 3203, 3221, 3223, 3321, 3323, 3331} and its isomorphic image under reflection and (01)(23).

45. (The maximum size of such a code is currently unknown. Algorithm C isn't fast enough to solve this problem on a single computer, but a sufficiently large cluster of machines and/or an improved algorithm should be able to discover the answer. The case $m = 3$ and $n = 6$ is also currently unsolved; a SAT solver shows quickly that a full set of $(3^6 - 3^3 - 3^2 + 3^1)/6 = 116$ codewords cannot be achieved.)

47. The 3-bit sequences 101, 111, 110 were rejected before seeing 000. In general, to make a uniformly random choice from q possibilities, the text suggests looking at the next $t = \lceil \lg q \rceil$ bits $b_1 \ldots b_t$. If $(b_1 \ldots b_t)_2 < q$, we use choice $(b_1 \ldots b_t)_2 + 1$; otherwise we reject $b_1 \ldots b_t$ and try again. [This simple method is optimum when $q \le 4$, and the best possible running time for other values of q uses more than half as many bits. But a better scheme is available for $q = 5$, using only $3\frac{1}{3}$ bits per choice instead of $4\frac{4}{5}$; and for $q = 6$, one random bit reduces to the case $q = 3$. See D. E. Knuth and A. C. Yao, *Algorithms and Complexity*, edited by J. F. Traub (Academic Press, 1976), 357–428, §2.]

Niho
CNF
SAT solver
unit clauses
uniformly random
reject
Knuth
Yao
complexity of calculation
computational complexity
Traub

48. It's the number of nodes on level $l+1$ (depth l) of the search tree. (Hence we can estimate the profile. Notice that $D = D_1 \ldots D_{l-1}$ in step E2 of Algorithm E.)

49. $Z_0 = C()$, $Z_{l+1} = c() + D_1 c(X_1) + D_1 D_2 c(X_1 X_2) + \cdots + D_1 \ldots D_l c(X_1 \ldots X_l) + D_1 \ldots D_{l+1} C(X_1 \ldots X_{l+1})$.

50. (a) True: The generating function is $z(z+1)\ldots(z+n-1)/n!$; see Eq. 1.2.10–(9).

(b) For instance, suppose $Y_1Y_2\ldots Y_l = 1457$ and $n = 9$. Alice's probability is $\frac{1}{1}\frac{1}{2}\frac{2}{3}\frac{1}{4}\frac{1}{5}\frac{5}{6}\frac{1}{7}\frac{7}{8}\frac{8}{9} = \frac{1}{3}\frac{1}{4}\frac{1}{6}\frac{1}{9}$. Elmo obtains $X_1X_2\ldots X_l = 7541$ with probability $\frac{1}{9}\frac{1}{6}\frac{1}{4}\frac{1}{3}$.

(c) The upper tail inequality (see exercise 1.2.10–22 with $\mu = H_n$) tells us that $\Pr(l \ge (\ln n)(\ln\ln n)) \le \exp(-(\ln n)(\ln\ln n)(\ln\ln\ln n) + O(\ln n)(\ln\ln n))$.

(d) If $k \le n/3$ we have $\sum_{j=0}^{k}\binom{n}{j} \le 2\binom{n}{k}$. By exercise 1.2.6–67, the number of nodes on the first $(\ln n)(\ln\ln n)$ levels is therefore at most $2\bigl(ne/((\ln n)(\ln\ln n))\bigr)^{(\ln n)(\ln\ln n)}$.

51. The key idea is to introduce recursive formulas analogous to (29):

$$m(x_1\ldots x_l) = c(x_1\ldots x_l) + \min\bigl(m(x_1\ldots x_l x_{l+1}^{(1)})d, \ldots, m(x_1\ldots x_l x_{l+1}^{(d)})d\bigr);$$
$$M(x_1\ldots x_l) = c(x_1\ldots x_l) + \max\bigl(M(x_1\ldots x_l x_{l+1}^{(1)})d, \ldots, M(x_1\ldots x_l x_{l+1}^{(d)})d\bigr);$$
$$\widehat{C}(x_1\ldots x_l) = c(x_1\ldots x_l)^2 + \sum_{i=1}^{d}\bigl(\widehat{C}(x_1\ldots x_l x_{l+1}^{(i)})d + 2c(x_1\ldots x_l)C(x_1\ldots x_l x_{l+1}^{(i)})\bigr).$$

They can be computed via auxiliary arrays `MIN`, `MAX`, `KIDS`, `COST`, and `CHAT` as follows: At the beginning of step B2, set `MIN[`l`]` $\leftarrow \infty$, `MAX[`l`]` $\leftarrow$ `KIDS[`l`]` $\leftarrow$ `COST[`l`]` $\leftarrow$ `CHAT[`l`]` $\leftarrow 0$. Set `KIDS[`l`]` $\leftarrow$ `KIDS[`l`]` $+1$ just before $l \leftarrow l+1$ in step B3.

At the beginning of step B5, set $m \leftarrow c(x_1\ldots x_{l-1}) +$ `KIDS[`l`]` $\times$ `MIN[`l`]`, $M \leftarrow c(x_1\ldots x_{l-1}) +$ `KIDS[`l`]` $\times$ `MAX[`l`]`, $C \leftarrow c(x_1\ldots x_{l-1}) +$ `COST[`l`]`, $\widehat{C} \leftarrow c(x_1\ldots x_{l-1})^2 +$ `KIDS[`l`]` $\times$ `CHAT[`l`]` $+ 2 \times$ `COST[`l`]`. Then, after $l \leftarrow l-1$ is positive, set `MIN[`l`]` $\leftarrow \min(m,$ `MIN[`l`]`$)$, `MAX[`l`]` $\leftarrow \max(M,$ `MAX[`l`]`$)$, `COST[`l`]` $\leftarrow$ `COST[`l`]` $+ C$, `CHAT[`l`]` $\leftarrow$ `CHAT[`l`]` $+ \widehat{C}$. But when l reaches zero in step B5, return the values m, M, C, $\widehat{C} - C^2$.

52. Let $p(i) = p_{X_1\ldots X_{l-1}}(y_i)$, and simply set $D \leftarrow D/p(I)$ instead of $D \leftarrow Dd$. Then node $x_1\ldots x_l$ is reached with probability $\Pi(x_1\ldots x_l) = p(x_1)p_{x_1}(x_2)\ldots p_{x_1\ldots x_l}(x_l)$, and $c(x_1\ldots x_l)$ has weight $1/\Pi(x_1\ldots x_l)$ in S; the proof of Theorem E goes through as before. Notice that $p(I)$ is the *a posteriori* probability of having taken branch I.

(The formulas of answer 51 should now use '$/p(i)$' instead of 'd'; and that algorithm should be modified appropriately, no longer needing the `KIDS` array.)

53. Let $p_{X_1\ldots X_{l-1}}(y_i) = C(x_1\ldots x_{l-1}y_i)/\bigl(C(x_1\ldots x_{l-1}) - c(x_1\ldots x_{l-1})\bigr)$. (Of course we generally need to know the cost of the tree before we know the exact values of these ideal probabilities, so we cannot achieve zero variance in practice. But the form of this solution shows what kinds of bias are likely to reduce the variance.)

55. The effects of lookahead, dynamic ordering, and reversible memory are all captured easily by a well-designed cost function at each node. But there's a fundamental difference in step C2, because different codeword classes can be selected for branching at the same node (that is, with the same ancestors $x_1\ldots x_{l-1}$) after C5 has undone the effects of a prior choice. The level l never surpasses $L+1$, but in fact the search tree involves hidden levels of branching that are implicitly combined into single nodes.

Thus it's best to view Algorithm C's search tree as a sequence of *binary* branches: Should x be one of the codewords or not? (At least this is true when the "max kill" strategy of answer 39 has selected the branching variable x. But if $r > 1$ and the poison list is empty, an r-way branch is reasonable (or an $(r+1)$-way branch when the slack is positive), because r will be reduced by 1 and the same class c will be chosen after x has been explored.)

profile
generating function
tail inequality
cumulative binomial distribution
recursive formulas
a posteriori
cost function
search tree
poison list
slack

If x has been selected because it kills many other potential codewords, we probably should bias the branch probability as in exercise 52, giving smaller weight to the "yes" branch because the branch that includes x is less likely to lead to a large subtree.

bias
Kullback
Leibler
divergence
generating function
Venn diagram

57. Let $p_k = 1/D^{(k)}$ be the probability that Algorithm E terminates at the kth leaf. Then $\sum_{k=1}^{M}(1/M)\lg(1/(Mp_k))$ is the Kullback–Leibler divergence $D(q\|p)$, where q is the uniform distribution (see exercise MPR–121). Hence $\frac{1}{M}\sum_{k=1}^{M}\lg D^{(k)} \ge \lg M$. (The result of this exercise is essentially true in *any* probability distribution.)

58. Let ∞ be any convenient value $\ge n$. When vertex v becomes part of the path we will perform a two-phase algorithm. The first phase identifies all "tarnished" vertices, whose DIST must change; these are the vertices u from which every path to t passes through v. It also forms a queue of "resource" vertices, which are untarnished but adjacent to tarnished ones. The second phase updates the DISTs of all tarnished vertices that are still connected to t. Each vertex has LINK and STAMP fields in addition to DIST.

For the first phase, set $d \leftarrow$ DIST(v), DIST$(v) \leftarrow \infty+1$, R $\leftarrow \Lambda$, T $\leftarrow v$, LINK$(v) \leftarrow \Lambda$, then do the following while T $\ne \Lambda$: $(*)$ Set $u \leftarrow$ T, T $\leftarrow$ S $\leftarrow \Lambda$. For each w — u, if DIST$(w) < d$ do nothing (this happens only when $u = v$); if DIST$(w) \ge \infty$ do nothing (w is gone or already known to be tarnished); if DIST$(w) = d$, make w a resource (see below); otherwise DIST$(w) = d+1$. If w has no neighbor at distance d, w is tarnished: Set LINK$(w) \leftarrow$ T, DIST$(w) \leftarrow \infty$, T $\leftarrow w$. Otherwise make w a resource (see below). Then set $u \leftarrow$ LINK(u), and return to $(*)$ if $u \ne \Lambda$.

The queue of resources will start at R. We will stamp each resource with v so that nothing is added twice to that queue. To make w a resource when DIST$(w) = d$, do the following (unless $u = v$ or STAMP$(w) = v$): Set STAMP$(w) \leftarrow v$; if R $= \Lambda$, set R $\leftarrow$ RT $\leftarrow w$; otherwise set LINK(RT) $\leftarrow w$ and RT $\leftarrow w$. To make w a resource when DIST$(w) = d+1$ and $u \ne v$ and STAMP$(w) \ne v$, put it first on stack S as follows: Set STAMP$(w) \leftarrow v$; if S $= \Lambda$, set S $\leftarrow$ SB $\leftarrow w$; otherwise set LINK$(w) \leftarrow$ S, S $\leftarrow w$.

Finally, when $u = \Lambda$, we append S to R: Nothing needs to be done if S $= \Lambda$. Otherwise, if R $= \Lambda$, set R $\leftarrow$ S and RT $\leftarrow$ SB; but if R $\ne \Lambda$, set LINK(RT) $\leftarrow$ S and RT $\leftarrow$ SB. (These shenanigans keep the resource queue in order by DIST.)

Phase 2 operates as follows: Nothing needs to be done if R $= \Lambda$. Otherwise we set LINK(RT) $\leftarrow \Lambda$, S $\leftarrow \Lambda$, and do the following while R $\ne \Lambda$ or S $\ne \Lambda$: (i) If S $= \Lambda$, set $d \leftarrow$ DIST(R). Otherwise set $u \leftarrow$ S, $d \leftarrow$ DIST(u), S $\leftarrow \Lambda$; while $u \ne \Lambda$, update the neighbors of u and set $u \leftarrow$ LINK(u). (ii) While R $\ne \Lambda$ and DIST(R) $= d$, set $u \leftarrow$ R, R $\leftarrow$ LINK(u), and update the neighbors of u. In both cases "update the neighbors of u" means to look at all w — u, and if DIST$(w) = \infty$ to set DIST$(w) \leftarrow d+1$, STAMP$(w) \leftarrow v$, LINK$(w) \leftarrow$ S, and S $\leftarrow w$. (It works!)

59. (a) Compute the generating function $g(z)$ (see exercise 7.1.4–209) and then $g'(1)$.

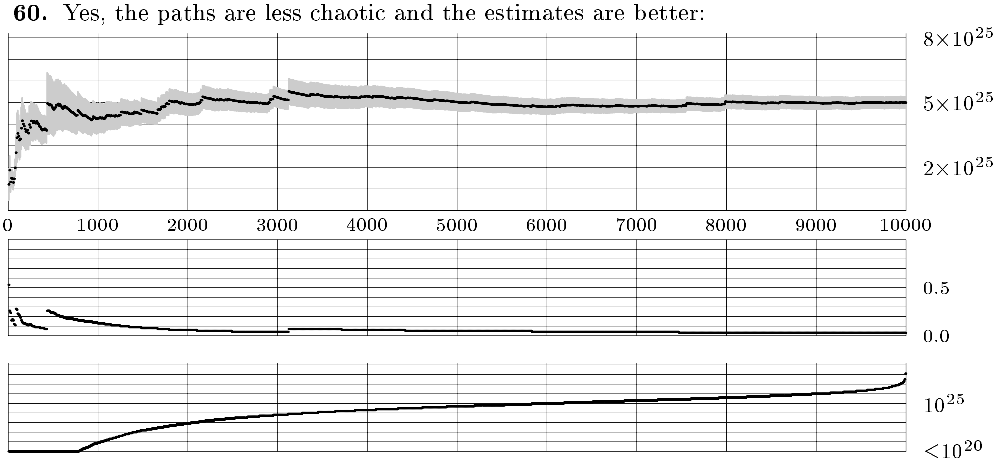
(b) Let (A, B, C) denote paths that touch (center, NE corner, SW corner). Recursively compute eight counts $(c_0, \ldots, c_7)$ at each node, where c_j counts paths π with $j = 4[\pi \in A] + 2[\pi \in B] + [\pi \in C]$. At the sink node $\boxed{\top}$ we have $c_0 = 1$, $c_1 = \cdots = c_7 = 0$. Other nodes have the form $x = (\bar{e}?\ x_l\colon x_h)$ where e is an edge. Two edges go across the center and affect A; three edges affect each of B and C. Say that those edges have types 4, 2, 1, respectively; other edges have type 0. Suppose the counts for x_l and x_h are $(c'_0, \ldots, c'_7)$ and $(c''_0, \ldots, c''_7)$, and e has type t. Then count c_j for node x is $c'_j + [t = 0]c''_j + [t \mathbin{\&} j \ne 0](c''_j + c''_{j-t})$.

(This procedure yields the following exact "Venn diagram" set counts at the root: $c_0 = |\overline{A} \cap \overline{B} \cap \overline{C}| = 7653685384889019648091604$; $c_1 = c_2 = |\overline{A} \cap \overline{B} \cap C| = |\overline{A} \cap B \cap \overline{C}| = 7755019053779199171839134$; $c_3 = |\overline{A} \cap B \cap C| = 7857706970503366819944024$;

$c_4 = |A \cap \overline{B} \cap \overline{C}| = 48885241665345737659950 71$; $c_5 = c_6 = |A \cap \overline{B} \cap C| = |A \cap B \cap \overline{C}| = 49493189917712521106051 48$; $c_7 = |A \cap B \cap C| = 50109501572837188079872 80$.)

student
grader
valid
dynamic ordering

60. Yes, the paths are less chaotic and the estimates are better:



90. Suppose there are n questions, whose answers each lie in a given set S. A *student* supplies an answer list $\alpha = a_1 \ldots a_n$, with each $a_j \in S$; a *grader* supplies a Boolean vector $\beta = x_1 \ldots x_n$. There is a Boolean function $f_{js}(\alpha, \beta)$ for each $j \in \{1, \ldots, n\}$ and each $s \in S$. A graded answer list (α, β) is *valid* if and only if $F(\alpha, \beta)$ is true, where

$$F(\alpha, \beta) = F(a_1 \ldots a_n, x_1 \ldots x_n) = \bigwedge_{j=1}^{n} \bigwedge_{s \in S} \bigl([a_j = s] \;\Rightarrow\; x_j \equiv f_{js}(\alpha, \beta)\bigr).$$

The *maximum score* is the largest value of $x_1 + \cdots + x_n$ over all graded answer lists (α, β) that are valid. A *perfect score* is achieved if and only if $F(\alpha, 1 \ldots 1)$ holds.

Thus, in the warmup problem we have $n = 2$, $S = \{\mathrm{A}, \mathrm{B}\}$; $f_{1\mathrm{A}} = [a_2 = \mathrm{B}]$; $f_{1\mathrm{B}} = [a_1 = \mathrm{A}]$; $f_{2\mathrm{A}} = x_1$; $f_{2\mathrm{B}} = \bar{x}_2 \oplus [a_1 = \mathrm{A}]$. The four possible answer lists are:

$$\begin{array}{ll}
\mathrm{AA}: & F = (x_1 \equiv [\mathrm{A} = \mathrm{B}]) \wedge (x_2 \equiv x_1) \\
\mathrm{AB}: & F = (x_1 \equiv [\mathrm{B} = \mathrm{B}]) \wedge (x_2 \equiv \bar{x}_2 \oplus [\mathrm{A} = \mathrm{A}]) \\
\mathrm{BA}: & F = (x_1 \equiv [\mathrm{B} = \mathrm{A}]) \wedge (x_2 \equiv x_1) \\
\mathrm{BB}: & F = (x_1 \equiv [\mathrm{B} = \mathrm{A}]) \wedge (x_2 \equiv \bar{x}_2 \oplus [\mathrm{B} = \mathrm{A}])
\end{array}$$

Thus AA and BA must be graded 00; AB can be graded either 10 or 11; and BB has no valid grading. Only AB can achieve the maximum score, 2; but 2 isn't guaranteed.

In Table 666 we have, for example, $f_{1\mathrm{C}} = [a_2 \neq \mathrm{A}] \wedge [a_3 = \mathrm{A}]$; $f_{4\mathrm{D}} = [a_1 = \mathrm{D}] \wedge [a_{15} = \mathrm{D}]$; $f_{12\mathrm{A}} = [\Sigma_\mathrm{A} - 1 = \Sigma_\mathrm{B}]$, where $\Sigma_s = \sum_{1 \le j \le 20} [a_j = s]$. It's amusing to note that $f_{14\mathrm{E}} = [\{\Sigma_\mathrm{A}, \ldots, \Sigma_\mathrm{E}\} = \{2, 3, 4, 5, 6\}]$.

The other cases are similar (although often more complicated) Boolean functions — except for 20D and 20E, which are discussed further in exercise 91.

Notice that an answer list that contains both 10E and 17E must be discarded: It can't be graded, because 10E says '$x_{10} \equiv \bar{x}_{17}$' while 17E says '$x_{17} \equiv x_{10}$'.

By suitable backtrack programming, we can prove first that no perfect score is possible. Indeed, if we consider the answers in the order (3, 15, 20, 19, 2, 1, 17, 10, 5, 4, 16, 11, 13, 14, 7, 18, 6, 8, 12, 9), many cases can quickly be ruled out. For example, suppose $a_3 = \mathrm{C}$. Then we must have $a_1 \neq a_2 \neq \cdots \neq a_{16} \neq a_{17} = a_{18} \neq a_{19} \neq a_{20}$, and early cutoffs are often possible. (We might reach a node where the remaining

propagation algorithm
search tree
author
recursively
quantified
fixed point
indeterminate

choices for answers 5, 6, 7, 8, 9 are respectively $\{C, D\}$, $\{A, C\}$, $\{B, D\}$, $\{A, B, E\}$, $\{B, C, D\}$, say. Then if answer 8 is forced to be B, answer 7 can only be D; hence answer 6 is also forced to be A. Also answer 9 can no longer be B.) An instructive little propagation algorithm will make such deductions nicely at every node of the search tree. On the other hand, difficult questions like 7, 8, 9, are best *not* handled with complicated mechanisms; it's better just to wait until all twenty answers have been tentatively selected, and to check such hard cases only when the checking is easy and fast. In this way the author's program showed the impossibility of a perfect score by exploring just 52859 nodes, after only 3.4 megamems of computation.

The next task was to try for score 19 by asserting that only x_j is false. This turned out to be impossible for $1 \le j \le 18$, based on very little computation whatsoever (especially, of course, when $j = 6$). The hardest case, $j = 15$, needed just 56 nodes and fewer than 5 kilomems. But then, ta da, three solutions were found: One for $j = 19$ (185 kilonodes, 11 megamems) and two for $j = 20$ (131 kilonodes, 8 megamems), namely

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
D	C	E	A	B	E	B	C	E	A	B	E	A	E	D	B	D	A	b	B	(i)
A	E	D	C	A	B	C	D	C	A	C	E	D	B	C	A	D	A	A	c	(ii)
D	C	E	A	B	A	D	C	D	A	E	D	A	E	D	B	D	B	E	e	(iii)

(The incorrect answers are shown here as lowercase letters. The first two solutions establish the truth of 20B and the falsity of 20E.)

91. Now there's only *one* list of answers with score ≥ 19, namely (iii). But that is paradoxical — because it claims 20E is false; hence the maximum score *cannot* be 19!

Paradoxical situations are indeed possible when the global function F of answer 90 is used recursively within one or more of the local functions f_{js}. Let's explore a bit of recursive territory by considering the following two-question, two-letter example:

1. (A) Answer 1 is incorrect. (B) Answer 2 is incorrect.
2. (A) Some answers can't be graded consistently. (B) No answers achieve a perfect score.

Here we have $f_{1A} = \bar{x}_1$; $f_{1B} = \bar{x}_2$; $f_{2A} = \exists a_1 \exists a_2 \forall x_1 \forall x_2 \neg F(a_1a_2, x_1x_2)$; $f_{2B} = \forall a_1 \forall a_2 \neg F(a_1a_2, 11)$. (Formulas quantified by $\exists a$ or $\forall a$ expand into $|S|$ terms, while $\exists x$ or $\forall x$ expand into two; for example, $\exists a \forall x\, g(a, x) = \bigl(g(A,0) \wedge g(A,1)\bigr) \vee \bigl(g(B,0) \wedge g(B,1)\bigr)$ when $S = \{A, B\}$.) Sometimes the expansion is undefined, because it has more than one "fixed point"; but in this case there's no problem because f_{2A} is true: Answer AA can't be graded, since 1A implies $x_1 \equiv \bar{x}_1$. Also f_{2B} is true, because both BA and BB imply $x_1 \equiv \bar{x}_2$. Thus we get the maximum score 1 with either BA or BB and grades 01.

On the other hand the simple one-question, one-letter questionnaire '**1.** (A) The maximum score is 1' has an *indeterminate* maximum score. For in this case $f_{1A} = F(A, 1)$. We find that if $F(A, 1) = 0$, only $(A, 0)$ is a valid grading, so the only possible score is 0; similarly, if $F(A, 1) = 1$, the only possible score is 1.

OK, suppose that the maximum score for the modified Table 666 is m. We know that $m < 19$; hence (iii) isn't a valid grading. It follows that 20E is true, which means that *every valid graded list of score m has x_{20} false.* And we can conclude that $m = 18$, because of the following two solutions (which are the only possibilities with 20C false):

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
B	A	d	A	B	E	D	C	D	A	E	D	A	E	D	E	D	B	E	c
A	E	D	C	A	B	C	D	C	A	C	E	D	B	a	C	D	A	A	c

But wait: If $m = 18$, we can score 18 with 20A true and two errors, using (say)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
D	e	D	A	B	E	D	e	C	A	E	D	A	E	D	B	D	C	C	A

or 47 other answer lists. This *contradicts* $m = 18$, because x_{20} is true.

End of story? No. This argument has implicitly been predicated on the assumption that 20D is false. What if m is indeterminate? Then a new solution arises

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
D	C	E	A	B	E	D	C	E	A	E	B	A	E	D	B	D	A	d	D

of score 19. With (iii) it yields $m = 19$! If m is determinate, we've shown that m cannot actually be defined consistently; but if m is indeterminate, it's definitely 19.

Question 20 was designed to create difficulties. [:-)]
— DONALD R. WOODS (2001)

95. (a) Let x_k be the number of nodes at distance $k-1$ from the root.

(b) Let $Q_n^{(m)} = P_n^{(1)} + \cdots + P_n^{(m)}$. Then we have the joint recurrence $P_1^{(m)} = 1$, $P_{n+1}^{(m)} = Q_n^{(2m)}$; in particular, $Q_1^{(m)} = m$. And for $n \geq 2$, we have $Q_n^{(m)} = \sum_{k=1}^{n} a_{nk}\binom{m}{k}$ for certain constants a_{nk} that can computed as follows: Set $t_k \leftarrow P_n^{(k)}$ for $1 \leq k \leq n$. Then for $k = 2, \ldots, n$ set $t_n \leftarrow t_n - t_{n-1}, \ldots, t_k \leftarrow t_k - t_{k-1}$. Finally $a_{nk} \leftarrow t_k$ for $1 \leq k \leq n$. For example, $a_{21} = a_{22} = 2$; $a_{31} = 6$, $a_{32} = 14$, $a_{33} = 8$. The numbers $P_n^{(m)}$ have $O(n^2 + n\log m)$ bits, so this method needs $O(n^5)$ bit operations to compute P_n.

(c) $P_n^{(m)}$ corresponds to random paths with $X_1 = m$, $D_k = 2X_k$, $X_{k+1} = \lceil 2U_kX_k\rceil$, where each U_k is an independent uniform deviate. Therefore $P_n^{(m)} = \mathrm{E}(D_1 \ldots D_{n-1})$ is the number of nodes on level n of an infinite tree. We have $X_{k+1} \geq 2^kU_1\ldots U_km$, by induction; hence $P_n^{(m)} \geq \mathrm{E}(2^{\binom{n}{2}}U_1^{n-2}U_2^{n-3}\ldots U_{n-2}^1 m^{n-1}) = 2^{\binom{n}{2}}m^{n-1}/(n-1)!$.

[M. Cook and M. Kleber have discussed similar sequences in *Electronic Journal of Combinatorics* **7** (2000), #R44. See also K. Mahler's asymptotic formula for binary partitions, in *J. London Math. Society* **15** (1940), 115–123, which shows that $\lg P_n = \binom{n}{2} - \lg(n-1)! + \binom{\lg n}{2} + O(1)$.]

999. ...

WOODS
recurrence
Cook
Kleber
Mahler

INDEX AND GLOSSARY

GOLDSMITH

He writes indexes to perfection.
— OLIVER GOLDSMITH, *Citizen of the World* (1762)

When an index entry refers to a page containing a relevant exercise, see also the *answer* to that exercise for further information. An answer page is not indexed here unless it refers to a topic not included in the statement of the exercise.

2-letter block codes, 27.
4-letter codewords, 9–18, 27–29.
5-letter words of English, 8–9, 27.
6-letter and k-letter words of English, 8–9.
γ (Euler's constant), as source of "random" data, 19.
π (circle ratio), as source of "random" data, 19–20, 22, 28.
ϕ (golden ratio), as source of "random" data, 19.

A posteriori probability, 39.
Active elements of a list, 12.
Ahrens, Joachim Heinrich Lüdecke, 6, 26, 33.
Analysis of algorithms, 28, 29.
Aperiodic words, 10, 27, 36.

Backjumping, 24.
Backmarking, 24.
Backtrack programming, 2–∞.
  efficiency of, 32.
  history of, 2, 5–6, 24–25.
  introduction to, 2–31.
Backtrack trees, 3, 4, 7, 9–11, 18–20, 38, 39, 41.
  estimating the size of, 20, 38.
Baumert, Leonard Daniel, 24, 45.
Bees, queen, 26.
Bernoulli, Jacques (= Jakob = James), 24.
Bezzel, Max Friedrich Wilhelm, 24.
Biased random walks, 29, 40.
Binary partitions, 31.
Binomial trees, 21.
Bitner, James Richard, 6, 24.
Bitwise operations, 5, 26, 34.
Block codes, 9, 27.
Boundary markers, 28.
Breadth-first search, 33.
Breaking symmetry, 8, 14.
British National Corpus, 8.
Broken diagonals, *see* Wraparound.
Bumping the current stamp, 16, 28.
Bunch, Steve Raymond, 6.

Cache-friendly data structures, 11.
Carroll, Lewis (= Dodgson, Charles Lutwidge), iii.
Cavenagh, Nicholas John, 33.
Cayley, Arthur, 31.
Cells of memory, 11.
Chatterjee, Sourav (সৌরভ চ্যাটার্জী), 22.
Chessboard, 2–6, 22–24, 26, 30.
Chiral symmetry, 26.
Closed lists, 14.
CNF: Conjunctive normal form, 38.
Codewords, commafree, 9–18, 27–29.
Coin flipping, 29.
Combinations, 26.
Commafree codes, 9–18, 24, 27–29.
Compilers, 15.
Complexity of calculation, 38.
Compressed tries, 35.
Computational complexity, 38.
Concatenation, 9.
Cook, Matthew Makonnen, 43.
Corner-to-corner paths, 22–23, 29–30.
Cost function, 19, 39.
Crick, Francis Harry Compton, 9.
Cumulative binomial distribution, 39.
Cutoff principle, 7.
Cutoff properties, 2, 5, 10, 18, 26.
Cyclic shifts, 10, 27.

Dancing links, 7.
Data structures, 4–6, 9, 11–14, 18, 28.
de Jaenisch, Carl Friedrich Andreevitch (Янишъ, Карлъ Андреевичъ), 32.
Degree of a node, 19.
Deletion operation, 7, 12–13, 37.
Depth-first search, 24.
Diaconis, Persi Warren, 22.
Diagonal lines (slope ± 1), 3, 33, *see also* Wraparound.
Digraphs, 27.
DIMACS: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, inaugurated in 1990.
Dips, 28.
Discarded data, 22.
Discrete probabilities, 1.
Distributed computations, 6.
Divergence, Kullback–Leibler, 40.
Dodgson, Charles Lutwidge, iii.
Domains, 2, 26, 27.
Downdating versus updating, 4–5, 11, 15.
Dual solutions, 6, 8, 27.
Dynamic ordering, 10–11, 24, 29, 41.

Dynamic shortest distances, 29.

e, as source of "random" data, 19.
Eastman, Willard Lawrence, 27, 28, 36.
Eight queens problem, 3–4, 19–20, 24.
Empty lists, 12, 17.
English words, 8–9, 27.
Error bars, 22.
Estimates of run time, 18–21.
Estimating the number of solutions, 21–23.

Fallback points, 16.
Five-letter words, 8–9, 27.
Fixed point of recursive formula, 42.
Floyd, Robert W, 15.
Four-letter codewords, 9–18, 27–29.
FPGA devices: Field-programmable gate arrays, 6.
Frames, 11.
Franel, Jérome, 33.

Gaschnig, John Gary, 24.
Gattegno, Bernard, 35.
Gauß (= Gauss), Johann Friderich Carl (= Carl Friedrich), 24, 32.
Generating functions, 28, 39, 40.
Gigamem (Gμ): One billion memory accesses, 5.
Global variables, 26.
Goldsmith, Oliver, 44.
Golomb, Solomon Wolf, 9, 24, 45.
Gordon, Basil, 9, 45.
Graders, 41.
Griffith, John Stanley, 9.

Hales, Alfred Washington, 45.
Hall, Marshall, Jr., 24.
Hamilton, William Rowan, paths, 23.
Hammersley, John Michael, 24.
Handscomb, David Christopher, 24.
Height of binary trees, 31.
Hexagons, 26.
Historical notes, 2, 5–6, 24–25.
Honeycombs, 26.
Hurwitz, Adolf, 33.

IBM 1620 computer, 6.
IBM System 360-75 computer, 6.
Importance sampling, 24.
Indeterminate statements, 42.
Inner loops, 32.
Insertion operation, 12.
Integer partitions, 26, 31.
Internet, ii, iii.
Inverse lists, 12–15, 17.
Inverse permutations, 12–13.
Iteration versus recursion, 26, 32.

Jaenisch, Carl Friedrich Andreevitch de (Янишъ, Карлъ Андреевичъ), 32.
Jewett, Robert Israel, 45.
Jiggs, B. H. (pen name of Baumert, Hales, Jewett, Imaginary, Golomb, Gordon, and Selfridge), 17.

Kennedy, Michael David, 6.
Kilomem (Kμ): One thousand memory accesses, 41.
King, Benjamin Franklin, Jr., 2.
King paths, 22–23, 25, 29–30.
Kleber, Michael Steven, 43.
Knight moves, 23.
Knuth, Donald Ervin (高德纳), i, iv, 18, 24, 26, 35, 38, 42.
Kullback, Solomon, 40.

Langford, Charles Dudley, pairs, 6–8, 26–27.
Laxdal, Albert Lee, 17.
Leibler, Richard Arthur, 40.
Lennon, John Winston Ono, 2.
Lexicographic order, 2, 7, 24, 28.
Linked lists, 6–7, 37.
Lookahead, 10, 16, 24, 29.
Loose Langford pairs, 27.
Lucas, François Édouard Anatole, 24, 33.

Mahler, Kurt, 43.
Martingales, 29.
Masks, 32.
Megamem (Mμ): One million memory accesses, 18.
`MEM`, an array of "cells," 11–18, 28.
Memory constraints, historic, 36.
Mem (μ): One 64-bit memory access, 4.
Minimum remaining values heuristic, 24.
`MMIX` computer, ii.
Monte Carlo estimates, 18–25, 29.
Moves, 11.
MPR: Mathematical Preliminaries Redux, v, 1.

n-letter words of English, 8.
n queens problem, 2–6, 18–20, 24, 26.
n-tuples, 26.
Nauck, Franz, 24.
Nested parentheses, 26.
Niho, Yoji Goff (仁保洋二), 37.
Nonisomorphic solutions, 29.

Onnen, Hendrick, Sr., 6.
Optimization, 24.
Orgel, Leslie Eleazer, 9.
Orthogonal lists, 27.
OSPD4: *Official SCRABBLE® Players Dictionary*, 8.
Overflow of memory, 12, 16.

$P_0()$, 2.
Paradox, 31.
Parentheses, 26.
Parity argument, 34.
Partitions, 26, 31.
Paths, simple, 22, 25, 29.
Pauls, Emil, 33.
Pencil-and-paper method, 18–20.
Periodic sequences, 27.
Periodic words, 10, 13.
Permutations, 6, 26.
Phi (ϕ), as source of "random" data, 19.
Pi (π), as source of "random" data, 19–20, 22, 28.
Poison list, 16–17, 28–29, 39.
Pólya, György (= George), 33.
Preußer, Thomas Bernd, 6.
Prime strings, 10.
Probabilities, 1.
Profile of a tree, 3, 9, 27, 31, 38.
Propagation algorithm, 42.
Properties: Logical propositions (relations), 2, 26.

q.s., 29.
Quantified Boolean formulas, 42.
Queen bees, 26.
Questionnaires, 30.
Queues, 36, 40.
Quick, Jonathan Horatio, 26.

Radix m representation, 13.
Random bits, 29.
Random sampling, 18.
Random variables, 19.
Random walks, 18–23, 29.
Recurrence relations, 39, 43.
  in a Boolean equation, 42.
Recursion versus iteration, 26, 32.
Recursive algorithms, 26, 39.
Reflection symmetry, 14, 26, 32, *see also* Dual solutions.
Registers, 5, 34–35.
Reingold, Edward Martin (ריינגולד, יצחק משה בן חיים), 24.
Rejection method, 19, 38.
Restricted growth strings, 32.
Reversible memory technique, 16, 29.
Rivin, Igor (Ривин, Игорь Евгеньевич), 33.
Root node, 19.
Rosenbluth, Arianna Wright, 25.
Rosenbluth, Marshall Nicholas, 25.
Rotation by 90°, 26.
Running time estimates, 18–21.

Sample variance, 22.
SAT solvers, 38.
Saturating ternary addition, 35.
Schumacher, Heinrich Christian, 24, 32.
Search rearrangement, *see* Dynamic ordering.
Search trees, 3, 4, 7, 9–11, 18–20, 33, 38, 39, 42.
  estimating the size, 20, 38.
Self-avoiding walks, 22, 25, 29.
Self-reference, 30, 46.
Self-synchronizing block codes, 9.
Selfridge, John Lewis, 45.
Semi-queens, 33.
Sequential allocation, 28.
Sequential lists, 11–15.
Set partitions, 26.
Shortest distances, dynamic, 29.
Signature of a trie node, 34.
Simple paths, 22, 25, 29.
Slack, 18, 39.
Sprague, Thomas Bond, 5, 6, 26.
Stacks, 16, 36.
Stamping, iv, 16–19, 28.
Standard deviation, 20, 22, 29.
Stanford GraphBase, ii, 8.
Statistics, 22.
Stirling, James, cycle numbers, 29.
Students, 41.
Substrings, 28.
Subtrees, 20.
Superdips, 36.
SWAC computer, 6.
Symmetries, 29, 32.
  breaking, 8, 14.

Tail inequality, 39.
Teramem (Tμ): One trillion memory accesses, 9.
Torus, 26.
Tot tibi . . . , 24.
Traub, Joseph Frederick, 38.
Trémaux, Charles Pierre, 24.
Tries, 8–9, 27, 34.
  compressed, 35.
Tuples, 26.
Twenty Questions, 30.
Two-letter block codes, 27.

UCLA: The University of California at Los Angeles, 6.
`UNDO` stack, 16.
Undoing, 4–5, 7, 15–16, 28.
Uniformly random numbers, 38.
Unit clauses, 38.
University of California, 6.
University of Dresden, 6.
University of Illinois, 6.
University of Tennessee, 6.
Unordered sequential lists, 11.
Unordered sets, 36.

Valid gradings, 41.
Vardi, Ilan, 33.
Variance of a random variable, 22, 29.
Venn, John, diagram, 40.
Visiting an object, 2, 4, 5, 7, 18.

Walker, Robert John, 2, 5, 6, 24.
Wanless, Ian Murray, 33.
Welch, Lloyd Richard, 9.
Wells, Mark Brimhall, 24.
White squares, 26.
Woods, Donald Roy, 30, 43.
Word rectangles, 8–9, 27.
WORDS(n), the n most common five-letter words of English, 8.
Worst-case bounds, 28.
Wraparound, 26.

Yao, Andrew Chi-Chih (姚期智), 38.

ZDD: A zero-suppressed decision diagram, 23, 30.
Zimmermann, Paul Vincent Marie, 33.